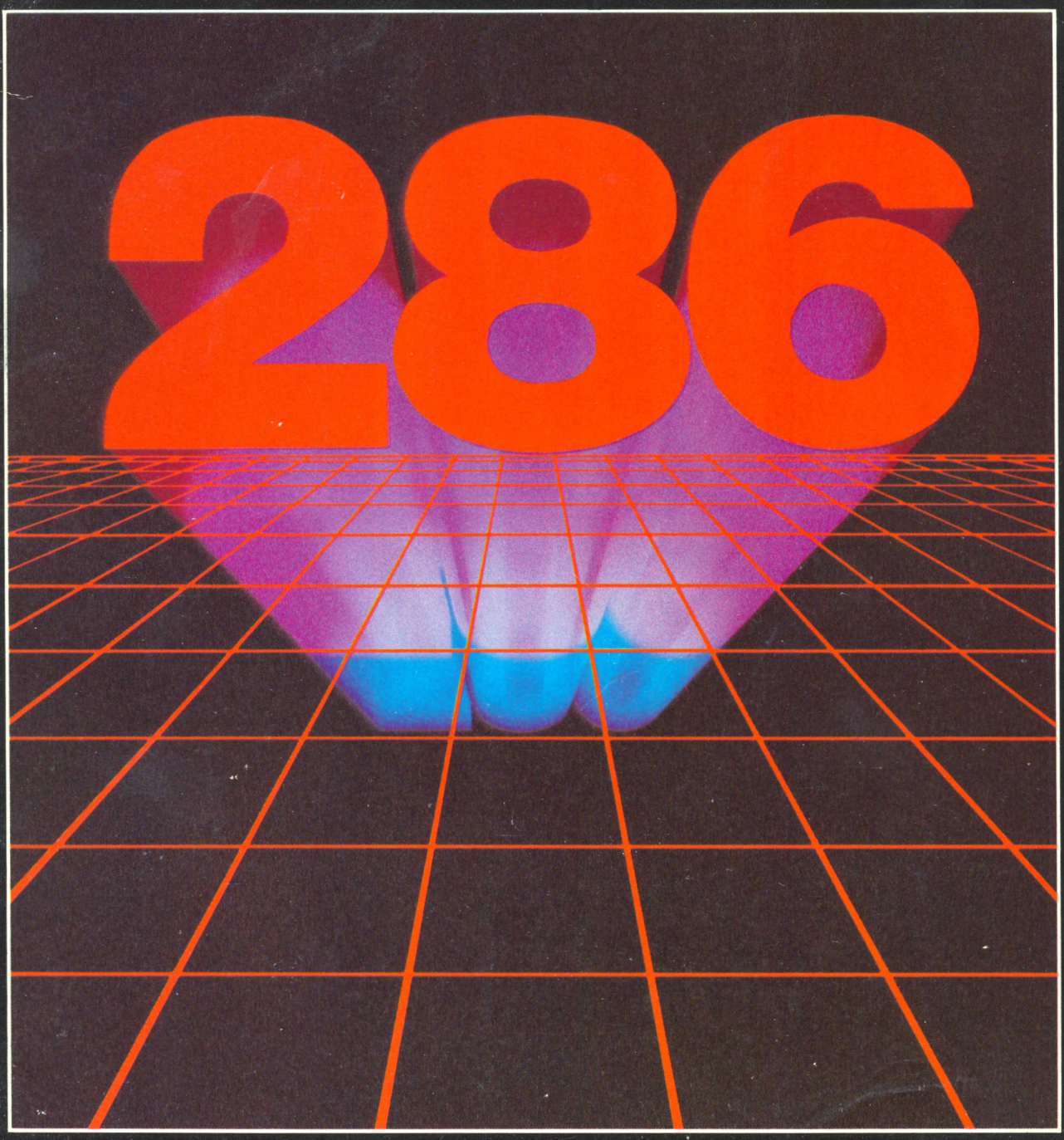
intel

iAPX 286 Operating Systems Writer's Guide



Order Number 121960-001

LITERATURE

1983 will be a year of transition for Intel's catalog program. In order to better serve you, our customers, we are reorganizing many of our catalogs to more completely reflect product groups.

In addition to the new product line handbooks listed below, an INTEL PRODUCT GUIDE (Order No. 210846) will be available free of charge in March. This GUIDE will contain a listing of Intel's complete product line along with information on quality/reliability, packaging and ordering, customer training classes and product services.

Consult the Intel Literature Guide (no charge, Order No. 210620) for a complete listing of Intel literature. Literature is presently available in other forms for those handbooks that will not be published until later in the year. Write or call the Intel Literature Department, 3065 Bowers Avenue, Santa Clara, CA 95051, (800) 538-1876, or (800) 672-1833 (California only).

HANDBOOKS

Memory Components Handbook (Order No. 210830)
Contains all application notes, article reprints, data sheets and other design information on RAMs, DRAMs, EPROMs, E^2PROMs, Bubble Memories.

Microcontroller Handbook (Available in May)
Contains all application notes, article reprints, data sheets, and other user information on the MCS-48, MCS-51 (8-bit) and the new MCS-96 (16-bit) product families.

Military Handbook (Order No. 210461)
Contains complete data sheets on all military products.

Microprocessor and Peripherals Handbook (Order No. 210844)
Contains data sheets on all microprocessors and peripherals. (Individual User Manuals are also available on the 8085, 8086, 8088, 186, 286, etc.)

Development Systems Handbook (Available in April)
Contains data sheets on development systems and supporting software.

OEM Systems Handbook (Available in May)
Contains all application notes, article reprints and data sheets for OEM boards and systems.

Software Handbook (Available in May)
Contains software product overview as well as data sheets for all Intel software.

Quality/Reliability Standards Handbook (Available in April)

iAPX 286 OPERATING SYSTEMS WRITER'S GUIDE

1983

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel Products:

BXP, CREDIT, i, ICE, I^2ICE, ICS, iDBP, iDIS, iLBX, i_m, iMMX, Insite, INTEL, int$_e$l, Intelevision, Intellec, int$_e$ligent Identifier™, int$_e$lBOS, int$_e$ligent Programming™, Intellink, iOSP, iPDS, iRMS, iSBC, iSBX, iSDM, iSXM, Library Manager, MCS, Megachassis, Micromainframe, MULTIBUS, Multichannel™ Plug-A-Bubble, MULTIMODULE, PROMPT, Ripplemode, RMX/80, RUPI, System 2000, and UPI, and the combination of ICE, iCS, iRMX, iSBC, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

* MULTIBUS is a patented Intel bus.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Department
3065 Bowers Avenue
Santa Clara, CA 95051

PREFACE

This book is written for systems designers and operating system designers and programmers who plan to use the Intel iAPX 286 microprocessor in its protected, virtual-address mode. The protected-mode iAPX 286 is designed for multitasking systems: systems that serve many users or that simultaneously run several programs.

This book analyzes operating system functions that are appropriate for the iAPX 286 when used in a variety of multitasking applications, including

- Communications, such as automated PBXs
- Real-time, such as instrumentation or process control
- Multi-user, such as time-sharing or office systems

Many of the features of the iAPX 286 are intended for use by an operating system. This book identifies and explains those features and gives examples of how they can be used in an operating system.

AUDIENCE

This book assumes that you have a knowledge of multitasking operating systems at least equivalent to that presented in introductory undergraduate textbooks on the subject. It also assumes that you have had some exposure to the architecture of the iAPX 286 through attending an introductory course or reading introductory literature such as *Introduction to the iAPX 286*.

RELATED PUBLICATIONS

Intel Literature

The following manuals contain additional information of use to operating-system designers and programmers:

- *ASM286 Assembly Language Reference Manual*, 121924
- *Component Data Catalog*, 210298
- *iAPX 286 Architecture Extension Kernel (K286) User's Guide*, 121961
- *iAPX 286 Programmer's Reference Manual*, 210498
- *iAPX 286 System Builder User's Guide*, 121935
- *iAPX 286 Utilities User's Guide*, 121934
- *iAPX 286/10 High Performance Microprocessor with Memory Management and Protection* (Data Sheet), 210253
- *Introduction to the iAPX 286*, 210308
- *PL/M-286 User's Guide*, 121945

External Literature

Many aspects of operating system construction for the iAPX 286 are the same as for other processors. The following are sources of generally applicable theories and algorithms referred to in the text of this book.

- Coffman, E.G., Jr., and Peter J. Denning, *Operating Systems Theory* (Englewood Cliffs, N.J.: Prentice-Hall, 1973)
- Denning, Peter J., "Virtual Memory," *Computing Surveys*, Vol. 2, No. 3 (September 1970)
- Knuth, Donald E., *Fundamental Algorithms*, Vol. 1 (Reading, Mass.: Addison-Wesley, 1973)
- Peterson, James L., *Petri Net Theory and the Modeling of Systems* (Englewood Cliffs, N.J.: Prentice-Hall, 1981)

RELATED PRODUCTS

Designers interested in operating systems for the protected-mode iAPX 286 should also be aware of Intel's iAPX 286 Architecture Extension Kernel K28. K286 is an operating-system kernel designed for a wide variety of applications, including real-time, communications, business systems, and time-sharing. K286 provides

- Short-term, priority scheduling and management of multiple tasks
- Interrupt management
- Multiprocessor support
- Virtual memory support
- Data sharing among tasks with synchronization
- Intertask signals and messages
- Extended protection

Whether you use K286 "as is," for greatest possible efficiency, or whether you add layers of software to more fully support your applications, K286 can significantly reduce your system development time. Since K286 has been designed by the architects of the iAPX 286 and implemented and tested by Intel's software engineers, using K286 can make your system more reliable.

K286 implements many of the concepts discussed in this book, which can therefore give you additional understanding of why and how to use K286.

HOW TO USE THIS MANUAL

This manual has two related objectives:

- To identify features of the iAPX 286 architecture that are unique when applied to the implementation of an operating system
- To show how you can effectively use these unique features in the design of familiar operating system functions

In pursuit of these objectives, Chapters 2 thru 13 share a general, three-part structure:

1. Identifing an operating system function (or class of functions). The functions chosen are those with which you might already be familiar, since they are similar to those used in state-of-the-art operating systems such as iRMX 86 (from Intel Corporation) or UNIX (from Bell Laboratories).
2. Reviewing the iAPX 286 features that support that function. While this portion of each chapter may cover some material available in other Intel literature, it provides added value by discussing in one place all the iAPX 286 features that bear on a given operating system function and by identifying relationships among those features.
3. Outlining some examples of how to use those iAPX 286 features in an implementation of the identified function. It is, of course, impossible to illustrate all the ways to design any given function, but these examples serve to illustrate a few of the ways that the designers of the iAPX 286 architecture intended for it to be applied.

Chapter 1 introduces the iAPX 286 architecture; identifies the role of an operating system in a protected, multitasking environment; and shows how Intel's Binder and Builder utilities aid in the construction of an operating system. You may skip Chapter 1 if you are already familiar with multitasking operating systems and with the Binder and the Builder.

Both Chapter 2 and Chapter 5 contain key information about manipulating the protection features of the iAPX 286. Be sure not to omit these chapters when scanning the contents of the book.

For the remaining chapters, you may turn directly to the subjects that interest you most. You will find the reading easier, however, if you observe the partial orderings outlined in table 0-1.

NOTATIONAL CONVENTIONS

UPPERCASE	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.
italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
Vx.y	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.

Table 0-1. Prerequisites by Chapter

Target Chapter	Prerequisites
6	4
7	6
8	4, 6
9	6, 7
12	4, 6, 7

Table of Contents

CHAPTER 1 **Page**

INTRODUCTION TO PROTECTED MULTITASKING ON THE iAPX 286

Tasks 1-1
Structure of a Program 1-1
Segmented Memory 1-1
Multitasking 1-1
Privilege Levels 1-3
Levels of Segments 1-4
Rules of Privilege 1-4
Software System Structure 1-4
Role of the Operating System 1-6
Common O.S. Functions 1-7
O.S. Functions in a Dynamic Environment 1-8
Constructing the Initial Run-Time Environment 1-8
Building a Static System 1-9
Building a Dynamic System 1-9

CHAPTER 2

USING HARDWARE PROTECTION FEATURES

Addressing Mechanism 2-1
Descriptors 2-2
Descriptor Format 2-2
Control Flow Transfer 2-7
Gate Descriptors 2-8
Control Transfer Mechanisms 2-9
Privilege Rules for Gated Intersegment Transfers 2-11
Descriptor Tables 2-12
Local Descriptor Table 2-14
Global Descriptor Table 2-14
Interrupt Descriptor Table 2-15
Selectors 2-15
Format of Selector 2-15
Null Selector 2-17
Alias Descriptors 2-17
Explicit Variation of Type 2-18
Variation of Length 2-18
Sharing Segments among Tasks 2-19
Protection and Integrity with Aliasing 2-19
Example of Descriptor Manipulation 2-20
Slot Management 2-22

CHAPTER 3 Page

REAL MEMORY MANAGEMENT

Memory Management Functions 3-1
Example of a Memory Manager 3-1
Data Structures 3-2
PL/M-286 Code 3-6
Protection Structure 3-7
Advanced Memory Management 3-10

CHAPTER 4

TASK MANAGEMENT

Hardware Task-Management Features 4-1
Storing Task State 4-1
Switching Tasks 4-3
Role of Operating System in Task Management 4-4
State Model of Task Scheduling 4-5
Interfacing with the Hardware Scheduler 4-6
Changing Scheduling State 4-7
Structuring Task Information 4-10
Example of a Dispatcher 4-13

CHAPTER 5

DATA SHARING, ALIASING, AND SYNCHRONIZATION

Data-Sharing Techniques 5-1
Sharing via the GDT 5-1
Sharing via Common LDT 5-1
Sharing via Aliases 5-2
Alias Management 5-3
Alias Database 5-3
Alias Procedures 5-3
Synchronization 5-4
Low-Level Mutual Exclusion 5-5
High-Level Mutual Exclusion 5-6
Message Passing 5-10
Message-Passing Example 5-10
Variations on the Mailbox Theme 5-16

CHAPTER 6

SIGNALS AND INTERRUPTS

Interrupt Features of the iAPX 286 Architecture 6-1
Vectoring 6-1
Enabling and Disabling Interrupts 6-2
Interrupt Descriptor Table 6-2
Interrupt Tasks and Interrupt Procedures 6-2

Page

Operating System Responsibilities 6-4
Manage IDT 6-5
Switch Scheduling Modes 6-5
Manage Interrupt Controller 6-6
Provide Task-Level Interrupt Procedures 6-7
Provide Software Signals 6-9

CHAPTER 7
HANDLING EXCEPTION CONDITIONS

Fault Mechanism 7-1
Fault Recovery 7-1
Locating the Faulting Instruction 7-1
Error Code 7-2
Application Interface 7-2
Exception Conditions 7-2
Interrupt 0—Divide Error 7-3
Interrupt 1—Single Step 7-3
Interrupt 3—Breakpoint 7-3
Interrupt 4—Overflow 7-3
Interrupt 5—Bound Check 7-4
Interrupt 6—Undefined Opcode (UD) 7-4
Interrupt 7—Processor Extension Not Available (NM) 7-4
Interrupt 8—Double Fault (DF) 7-5
Interrupt 9—Processor Extension Segment Overrun 7-5
Interrupt 10—Invalid TSS (TS) 7-5
Interrupt 11—Segment Not Present (NP) 7-6
Interrupt 12—Stack Exception (SS) 7-6
Interrupt 13—General Protection Exception (GP) 7-7
Interrupt 16—Processor Extension Error (MF) 7-8
Interrupt 17—Run-Time Exceptions 7-8
Restartability Summary 7-8

CHAPTER 8
INPUT/OUTPUT

I/O and Protection 8-1
I/O Privilege Level (IOPL) 8-1
Controlling I/O Addresses 8-2
I/O and Memory Management 8-3
Partitioning I/O Functions 8-3
Requirements for Parallelism and Synchronization 8-4
Requirements for Protection 8-4
Implementation Alternatives 8-5

CHAPTER 9 Page
VIRTUAL MEMORY
Hardware Mechanisms 9-1
Accessed Bit 9-1
Present Bit 9-1
Software Mechanisms 9-2
Secondary Storage Management 9-2
Level Zero Support Procedures 9-3
Swapping Managers 9-4
Software Policies 9-6
Fetch 9-6
Placement 9-6
Replacement 9-7
Thrashing 9-8

CHAPTER 10
SYSTEM INITIALIZATION
Initial State 10-1
Switching to Protected Mode 10-1
Initializing for Protected Mode 10-2
Interrupt Vector 10-2
Stack 10-2
Global Descriptor Table 10-2
Starting First Task 10-2
Example of Initialization 10-3
Initialization Module ENTP 10-3

CHAPTER 11
BINDING AND LOADING
Binding Model 11-1
Modules 11-1
Segmentation 11-2
Interfaces 11-2
Naming 11-3
Timing 11-3
Implementing According to the Model 11-3
Source Code 11-4
Compilers 11-4
Binding Utilities 11-5
Overview of Loading 11-8
Converting a Program into a Task 11-9
iAPX 286 Object Module Format 11-10
Flow of Loader 11-11
Load-Time Binding 11-12
Example Loader 11-12

CHAPTER 12
NUMERICS PROCESSOR EXTENSION Page

iAPX 286/20 Numerics Processing Features 12-1
ESCAPE Instructions 12-1
Emulation Mode Flag (EM) 12-1
Math Present Flag (MP) 12-2
Task Switched Flag (TS) 12-2
WAIT Instruction 12-2
Summary 12-2
Initialization 12-2
Task State 12-3
Numerics Exceptions 12-3
Interrupt 7—Processor Extension Not Available (NM) 12-3
Interrupt 9—Processor Extension Segment Overrun (MP) 12-4
Interrupt 16—Processor Extension Error (MF) 12-5

CHAPTER 13
EXTENDED PROTECTION

Extended Type 13-1
Type Extension Code with Descriptor 13-1
Type Extension Code in Segment 13-1
Indirect Naming 13-2
Parameter Validation 13-2
Defensive Use of ARPL 13-2
Point-of-Entry Scrutiny 13-3
Usage Privilege Level 13-4
Send Privilege Level 13-5
Constructing Shared Objects 13-5

GLOSSARY

INDEX

LIST OF TABLES

Table	Title	Page
0-1	Prerequisites by Chapter	v
2-1	Categories of Control Flow Transfer	2-7
2-2	Control Transfer Mechanisms	2-10
3-1	Actions for Combining Segments	3-9
4-1	Task Switching Operations	4-3
6-1	Interrupt Response Time	6-5
7-1	Restart Conditions	7-8
12-1	Interpretation of MP and EM Flags	12-3
13-1	Access Checking Instructions	13-2

LIST OF FIGURES

Figure	Title	Page
1-1	Segregation of Segments by Tasks with Private LDTs	1-2
1-2	Global Segments in System	1-3
1-3	Segment Segregation by Privilege Level within a Task	1-4
1-4	A Four-Level Protection Structure	1-6
1-5	A Two-Level Protection Structure	1-7
1-6	A One-Level Unprotected Structure	1-8
1-7	Independent Operating-System Tasks	1-9
1-8	Building a Static System	1-10
1-9	Building a Dynamic System	1-11
2-1	Abstraction of Addressing Mechanism	2-1
2-2	Data Segment Descriptor	2-3
2-3	Executable Segment Descriptor	2-4
2-4	System Segment Descriptor	2-5
2-5	Calling a Conforming Segment	2-6
2-6	Gate Descriptor	2-9
2-7	Intralevel Control Transfers	2-11
2-8	Gated Interlevel Call and Return	2-12
2-9	Valid and Invalid Interlevel Transfers	2-13
2-10	Format of a Selector	2-16
2-11	Aliasing for Debugger	2-17
2-12	Aliases for System Tables	2-18
2-13	Aliases with Differing Limit	2-19
2-14	Application of Segment Sharing	2-20
2-15	Aliases for Segment Sharing	2-21
2-16	Descriptor Manipulation Example	2-23
2-17	Available Slot List	2-26
3-1	Memory Fragmentation	3-2
3-2	Information Hiding in Memory-Management Example	3-3
3-3	Example Memory-Management Data Structures	3-4
3-4	Using Boundary Tags	3-5
3-5	Hiding Boundary Tags	3-6
3-6	Example GDT Layout	3-7
3-7	Splitting an Available Block of Memory	3-8
3-8	Possibilities for Combining Segments	3-11
3-9	Code for Memory-Management Example	3-15
4-1	Task State Segment and Register	4-2
4-2	Scheduling State Transition Diagram	4-5
4-3	Expanded Scheduling State Transition Diagram	4-6
4-4	Changing Scheduling Mode	4-8
4-5	Task Information Structure A	4-11
4-6	Task Information Structure B	4-12

Figure	Title	Page
4-7	Task Information Structure C	4-13
4-8	Scheduler Queue Segment	4-14
4-9	Dispatcher Example	4-15
5-1	Segment via Common LDT	5-2
5-2	Alias List	5-4
5-3	Identifying Alias List	5-5
5-4	Semaphore Structure	5-6
5-5	Semaphore Example	5-8
5-6	Mailbox Structure	5-11
5-7	Example of Mailbox Procedures	5-13
6-1	Interrupt Vectoring for Tasks	6-3
6-2	Interrupt Vectoring for Procedures	6-3
6-3	Interrupt Procedure's Stack	6-4
6-4	Private Interrupt Procedure	6-8
7-1	Exception Error Code	7-2
8-1	Petri Net Graph Symbols	8-5
8-2	Synchronization with Simple Device Driver	8-6
8-3	Synchronization with Two-Part Device Driver	8-6
10-1	Initialization Module ENTP	10-4
10-2	Dummy Segments for ENTP	10-14
10-3	Initial Task INIT	10-15
11-1	Subsystem for Kernel Exports	11-5
11-2	Binder Specifications for XOS Kernel	11-6
11-3	Builder Specifications for XOS	11-7
11-4	Specifying Dummy Gate Exports	11-13
11-5	Strategy for Load-Time Binding	11-14
11-6	Binding Loader	11-15
11-7	BOND Module of Binding Loader	11-24
13-1	Caller's CPL	13-3

Introduction to Protected Multitasking on the iAPX 286

1

1 Introduction to Protected Multitasking on the iAPX 286

CHAPTER 1
INTRODUCTION TO PROTECTED MULTITASKING ON THE iAPX 286

The iAPX 286 architecture views the software system (the operating system as well as applications) as a number of asynchronous tasks, each task possibly consisting of levels of procedures deserving different degrees of "trust." An operating system for the iAPX 286 must (with the help of the hardware) coordinate the activities of multiple tasks and administer protection among tasks and among levels of procedures within tasks.

TASKS

A task is the execution of a sequence of steps. A program is a logical entity that can have many representations: for example, source code file or object program file. A program becomes a task when it is actually available for execution. This is achieved by converting source (with a compiler and program loader, for example) to a representation suitable for execution and notifying the operating system that the task is ready for installation and execution.

The distinction between programs and tasks is most clear in a multitasking system, where it is possible for two or more tasks to use one program simultaneously. The line editor program in a timesharing system is a common example. Even though each line editor task uses the same program, each task produces different results, since it receives different inputs.

Structure of a Program

Each program is formed from modules created by language translators and bound together into a single executable unit. The translators (for example, ASM286, PL/M-286, Pascal-286, and FORTRAN-286) and the object program utilities (for example, Intel's Builder and Binder) support the concept of *logical segments*. A logical segment is a contiguous block of either instructions or data. Each logical segment can contain up to 64K bytes of code or data. Logical segments are the units that may be combined when a program comprises more than one module.

Segmented Memory

The iAPX 286 structures the address space of a task into *physical segments* of variable length. A physical segment is a contiguous block of memory that does not (normally) overlap another physical segment. Each physical segment may contain up to 64K bytes of either instructions or data. Each physical segment contains one or more logical segments of a task. The segments reflect how tasks are organized into code, data, and stack areas.

Multitasking

One of the most important features of the iAPX 286 is its ability to switch rapidly from executing one task to executing another task. This gives the appearance that the processor is executing more than one task at a time.

Hardware system tables enable both the hardware and the operating system to distinguish between the physical segments of individual tasks. Figure 1-1 shows how physical segments of one task are logically separate from those of other tasks. Since references to physical segments are always relative to system descriptor tables, the actual locations of physical segments in physical memory are not significant to the tasks and therefore are not illustrated.

Descriptor tables serve not only to identify the segments that belong to a task but also to *isolate* the address space of one task from that of another, so that one task cannot inadvertently affect the operations of another.

Multitasking works through close interaction of the operating system with hardware features. When the executing task needs to wait for some event (such as the arrival of data from some I/O device), it notifies the operating system. The operating system determines which other task should execute next, and then causes the processor to store the state of the current task, retrieve the state of the next task, and begin executing the next task at the point where its processing last halted. The processor then executes that task until the task needs to wait for some event. (This is a somewhat oversimplified

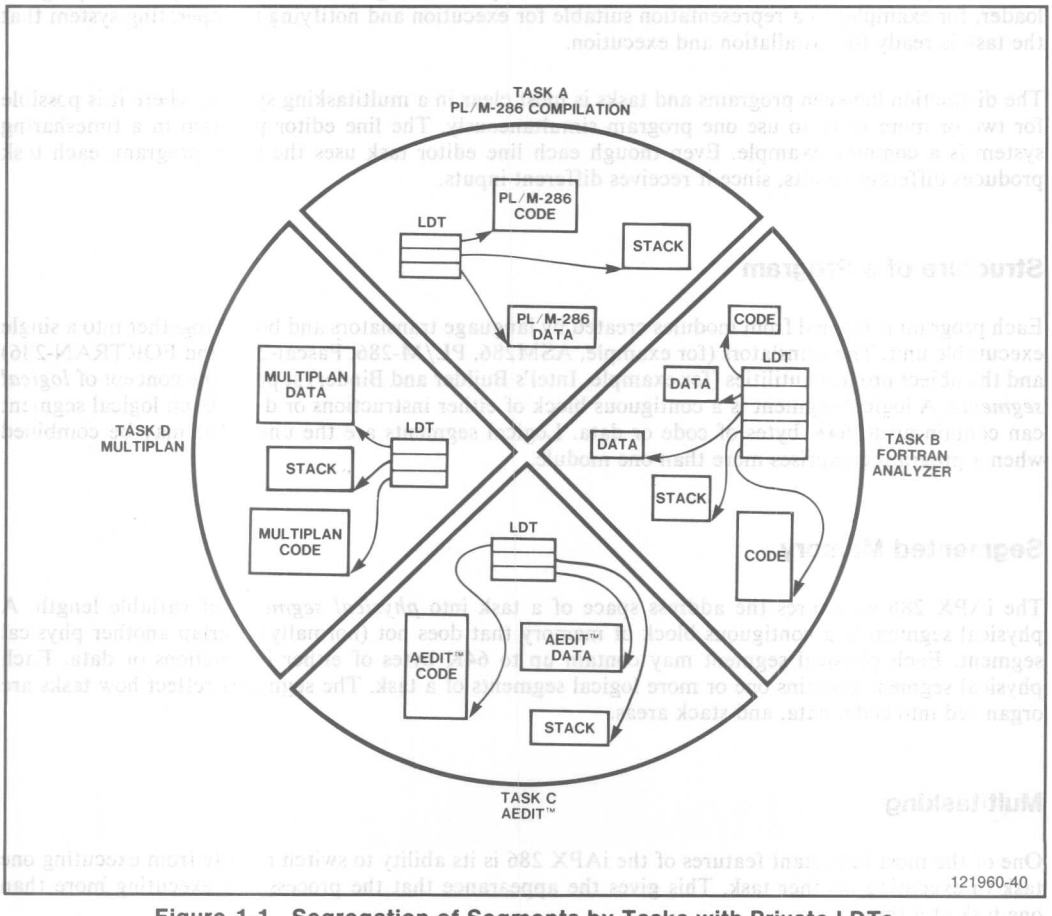


Figure 1-1. Segregation of Segments by Tasks with Private LDTs

description of what can be a complex operating system function. Chapter 4 covers the subject of task switching in more detail.)

Figure 1-2 illustrates how the global descriptor table defines an address space that is accessible to all tasks in the system. This global space is useful for translation tables, run-time libraries, operating-system code and data, and the like.

PRIVILEGE LEVELS

The iAPX 286 architecture uses the concept of *privilege levels* to protect critical procedures within a task from less trusted procedures in the same task. For example, with previous generations of microprocessors, applications code could access and possibly destroy tables used by the operating system. An error of this sort could cause the operating system to incorrectly service a subsequent request from another unrelated task. With the iAPX 286, such situations are prevented by hardware-enforced barriers between different levels of procedures.

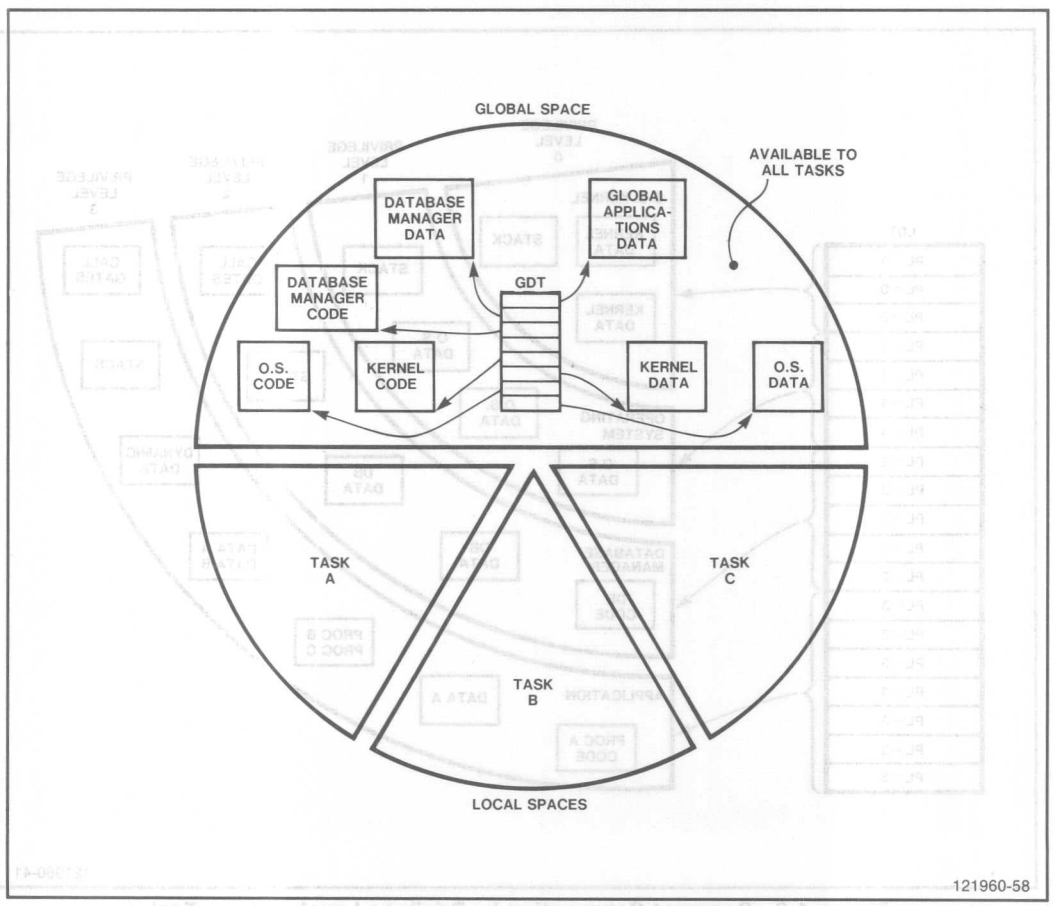


Figure 1-2. Global Segments in System

Applied to procedures, privilege level is a measure of the degree to which you trust a procedure not to make a mistake that might affect other procedures or data. Applied to data, privilege level is a measure of the protection you think a data structure should have from less trusted procedures.

Privilege level also applies to instructions. Certain instructions (those that deal with system tables, interrupts, and I/O, for example) have such an effect on the system as a whole that only highly trusted procedures should have the right to use them.

Levels of Segments

With regard to privilege, you can view the segments of a task as being grouped into four levels. Level zero is for the most privileged procedures and the most private data; level three is for the least trusted procedures and the most public data. You (or your operating system) associate each segment of each task with one of these four levels of privilege. The privilege level of a segment applies to all the procedures or all the data in that segment. Figure 1-3 illustrates the logical segregation of segments into privilege levels. (Later chapters explain why operating-system segments are included within the task.)

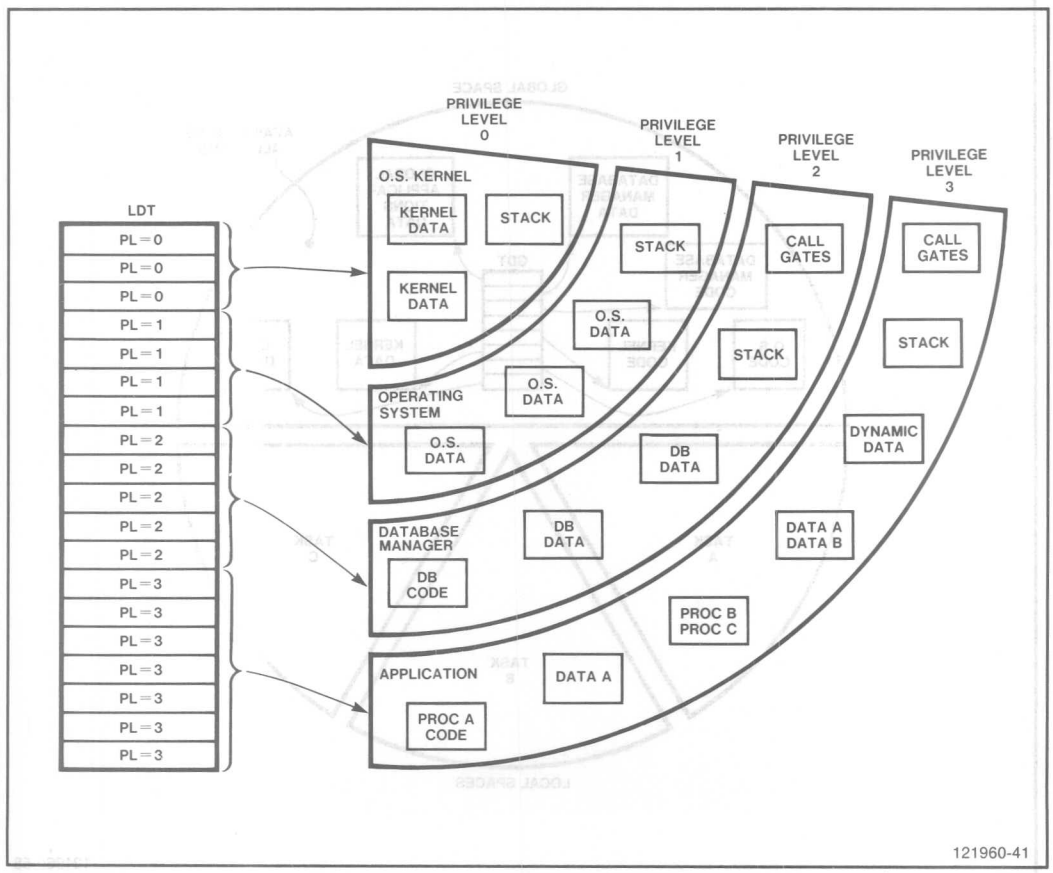


Figure 1-3. Segment Segregation by Privilege Level within a Task

Rules of Privilege

The 80286 processor controls access to both data and procedures between levels of a task. These rules define access rights:

- Data can be accessed only from the same or a more privileged level.
- A procedure can be called only from the same level (or from a less privileged level, if the service is deliberately "exported" to that level. Refer to gates in Chapter 2.).

SOFTWARE SYSTEM STRUCTURE

The way you choose to distribute software and data among tasks and privilege levels affects the reliability, efficiency, and flexibility of your system. Operating-system modules may be segregated into their own tasks or may be distributed among and shared by every task. Some advantages of placing operating-system modules in separate tasks are

- Finer granularity of protection is achieved by using task separation as well as privilege levels.
- Operating-system functions can execute in parallel with the caller.
- When only one task at a time can perform the function (for example, reading from a keyboard), serialization of requests is automatic; you do not need to synchronize among requesting tasks.

Some advantages of distributing operating-system functions are

- The communication between application and operating system is faster.
- It may be possible for all tasks to execute the same operation-system function in parallel. (You must ensure reentrancy and provide for synchronization, however.)

Figures 1-4 through 1-7 illustrate some general approaches that you may consider.

The approach shown in figure 1-4 takes maximum advantage of the four privilege levels. The critical procedures and data of the operating system kernel (for example, memory allocation tables and procedures, information about tasks) are protected from all other procedures in the system. Those parts of the operating system that are less reliable, either due to inherent complexity (for example, the I/O subsystem) or due to occasional changes (for example, policies designed to increase overall throughput), are at a lower level but are still protected from application levels. Application logic has two levels so that application services (such as database management) can be protected from less trusted application code, yet application services cannot affect the integrity of the operating system. Operating-system procedures and application services are shared among all the tasks in the system.

Figure 1-5 illustrates that you do not need to use all four privilege levels. You may prefer this two-level approach if you are converting from a traditional multitasking system that offers protection only between the two levels defined by application and operating system.

The iAPX 286 can also emulate one-level systems, as illustrated in figure 1-6. This approach may be useful in the initial stages of converting from an unprotected system, but it does not take advantage of many of the protection features offered by the iAPX 286. It does isolate tasks from one another, but it does not protect the operating system from applications software.

Some operating system functions are better structured as independent operating-system tasks, not as privileged procedures within a task. Certain I/O functions are suited to this treatment. Because of the complexity of I/O code, the extra protection offered by a task boundary contributes to the reliability

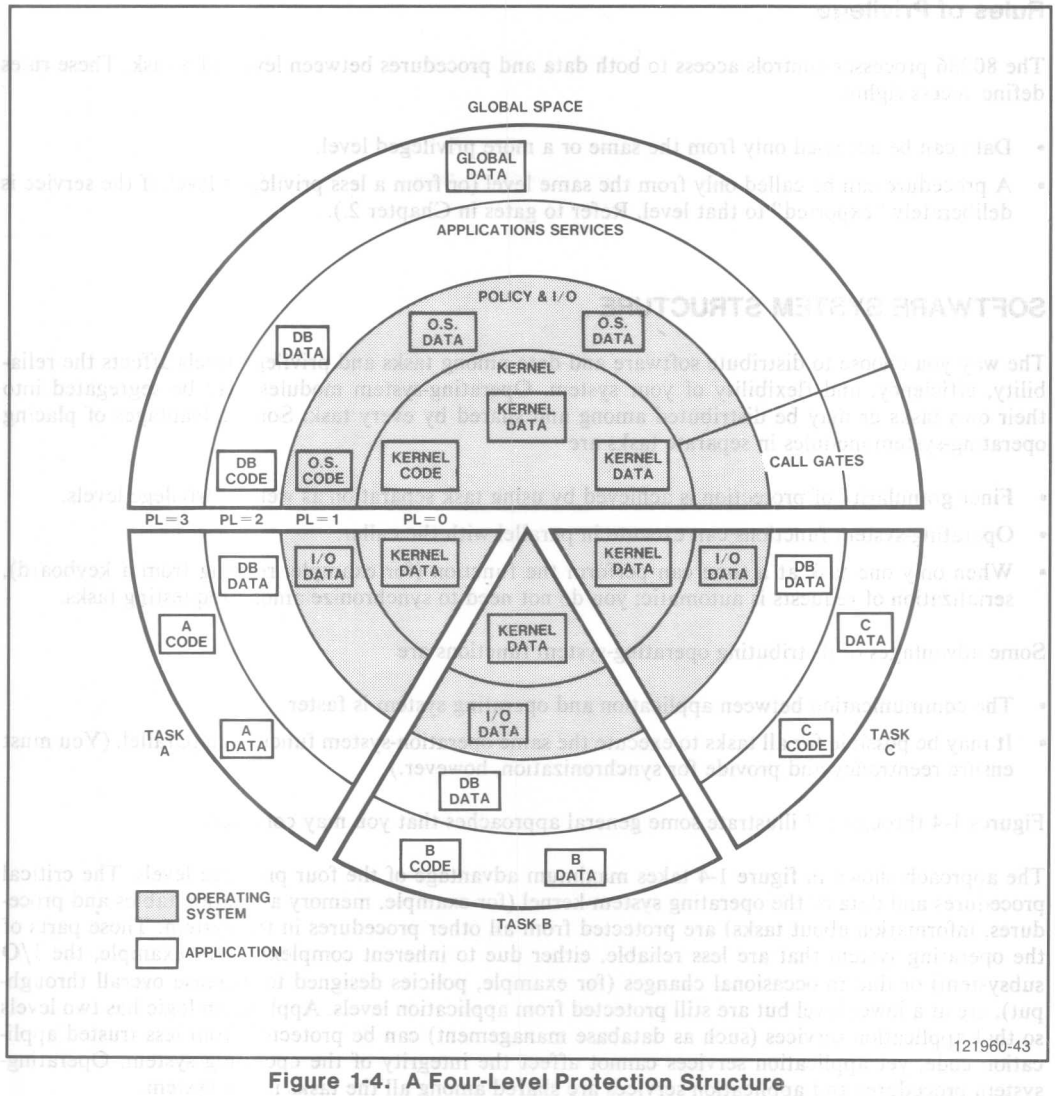


Figure 1-4. A Four-Level Protection Structure

of the system. Because many I/O functions involve waiting for responses from I/O devices, it is most convenient to treat these functions as a separate task that can run asynchronously with respect to the tasks that invoke them. Figure 1-7 illustrates a structure with independent operating-system tasks.

ROLE OF THE OPERATING SYSTEM

The role that an operating system may play in managing a multitasking execution environment depends on the nature of the application. Applications can be classified according to the volatility of tasks

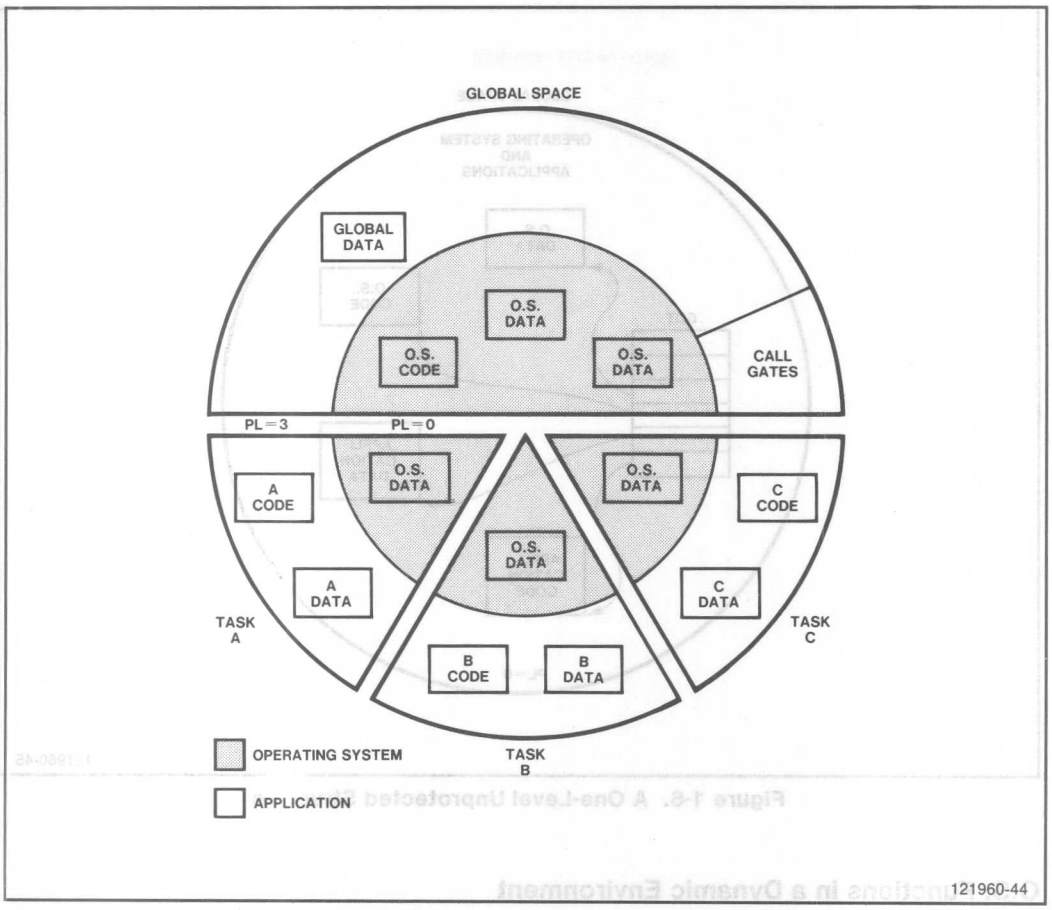


Figure 1-5. A Two-Level Protection Structure

executing over a period of time. Applications in which tasks frequently begin and end (for example, time-sharing systems or multi-user business systems) are called *dynamic systems*. Applications in which the mix of tasks does not change (for example, process control systems in which tasks service a fixed number of sensors and effectors) are called *static systems*.

Common O.S. Functions

The operating-system roles common to both static and dynamic applications are

- Allocation of the processor or processors to tasks
- Coordination and communication among cooperating tasks
- Processing of interrupts and exception conditions
- Standardization of interfaces to I/O devices
- Control of the numerics processor extension

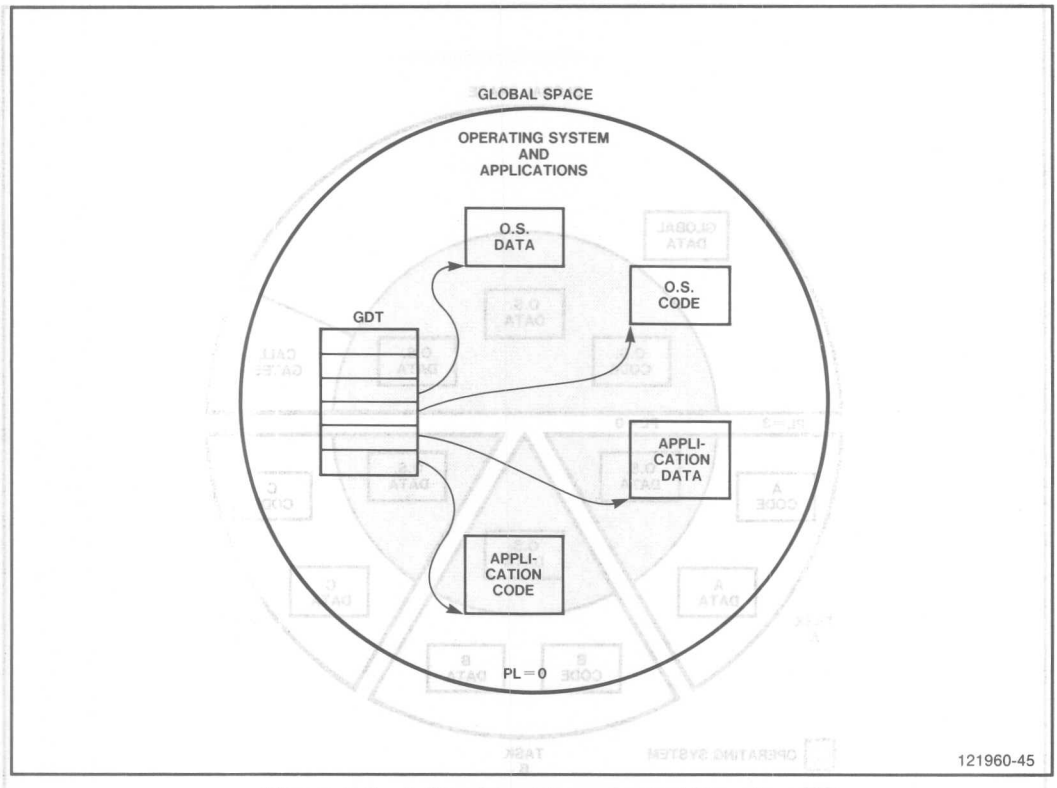


Figure 1-6. A One-Level Unprotected Structure

O.S. Functions in a Dynamic Environment

Even though many of the duties of an operating system in a dynamic environment resemble those in a static environment, the dynamic environment often introduces new complexities. Some additional functions that a dynamic system may require include

- Real memory management
- Program loading
- Command language interface
- Virtual memory management
- Load-time binding

CONSTRUCTING THE INITIAL RUN-TIME ENVIRONMENT

Intel's System Builder program helps you create the initial executable system. The Builder program collects object modules into one module, assigns physical addresses, creates system tables, and assigns privilege levels. A specification language gives you the ability to control precisely what the Builder does.

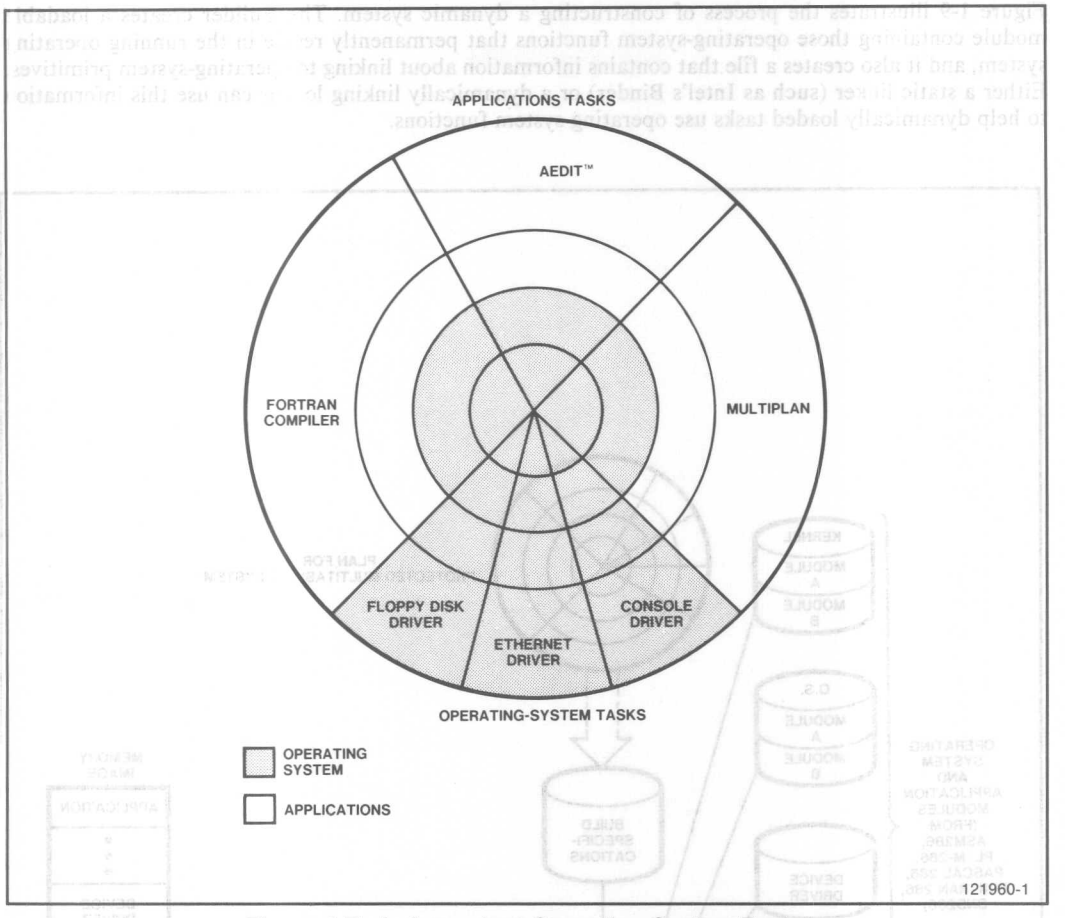


Figure 1-7. Independent Operating-System Tasks

Building a Static System

In the case of static systems, the Builder does nearly all the work in constructing a running system. Refer to figure 1-8 for an illustration of the building process. The output of the Builder is a single module that contains all the tasks, both for the operating system and for applications, as well as all system tables and protection information. The Builder's output file has a format that simplifies creation of a bootstrap loader for the system.

Building a Dynamic System

With dynamic systems, the Builder constructs as much of the final system as you can specify in advance, but the nature of dynamic systems is such that the operating system must do at run-time many of the operations that the Builder does for static systems (for example, allocation of memory and assignment of physical addresses). The operating system must update system tables and administer the protection mechanisms as the running environment changes.

Figure 1-9 illustrates the process of constructing a dynamic system. The Builder creates a loadable module containing those operating-system functions that permanently reside in the running operating system, and it also creates a file that contains information about linking to operating-system primitives. Either a static linker (such as Intel's Binder) or a dynamically linking loader can use this information to help dynamically loaded tasks use operating-system functions.

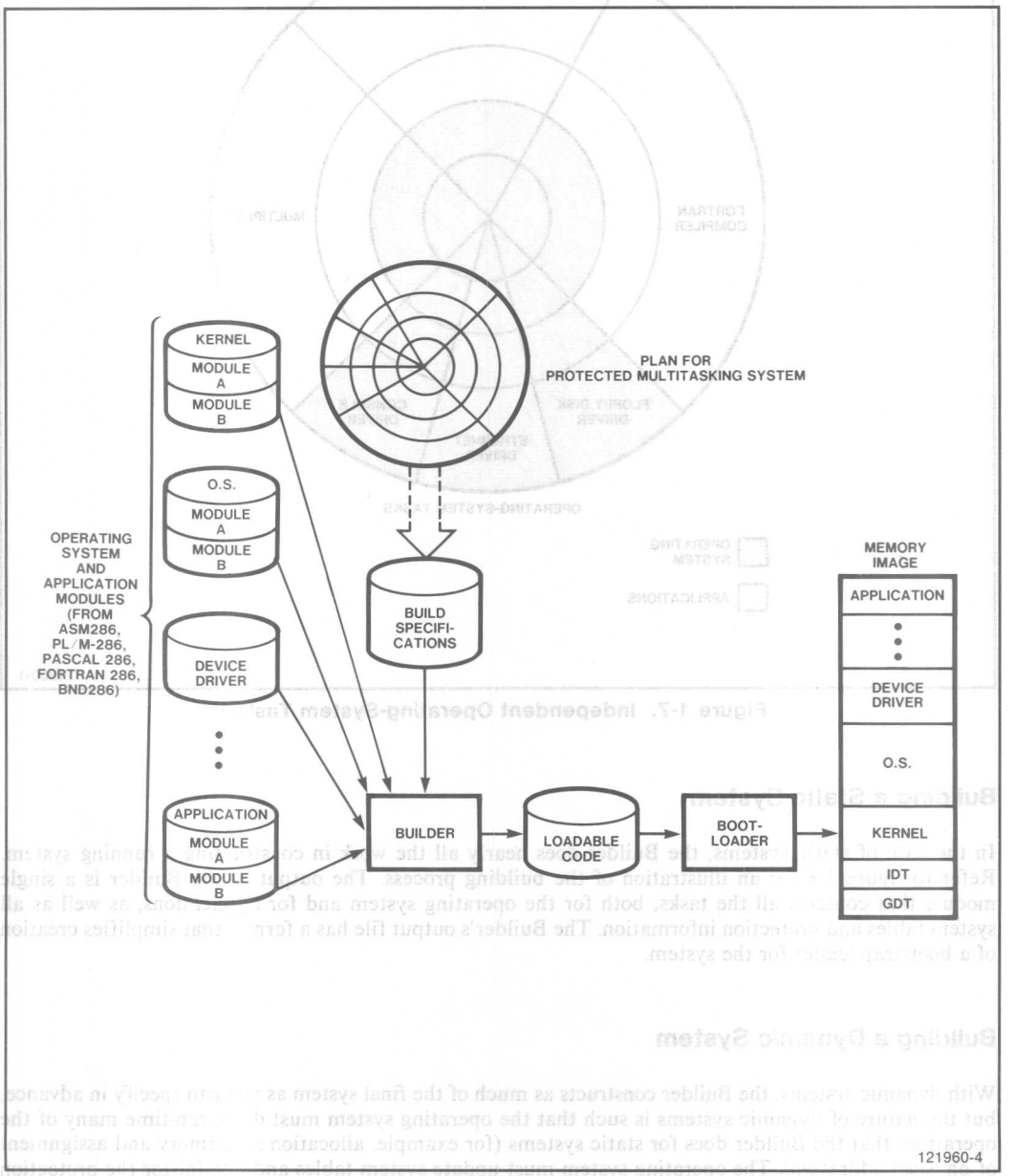


Figure 1-8. Building a Static System

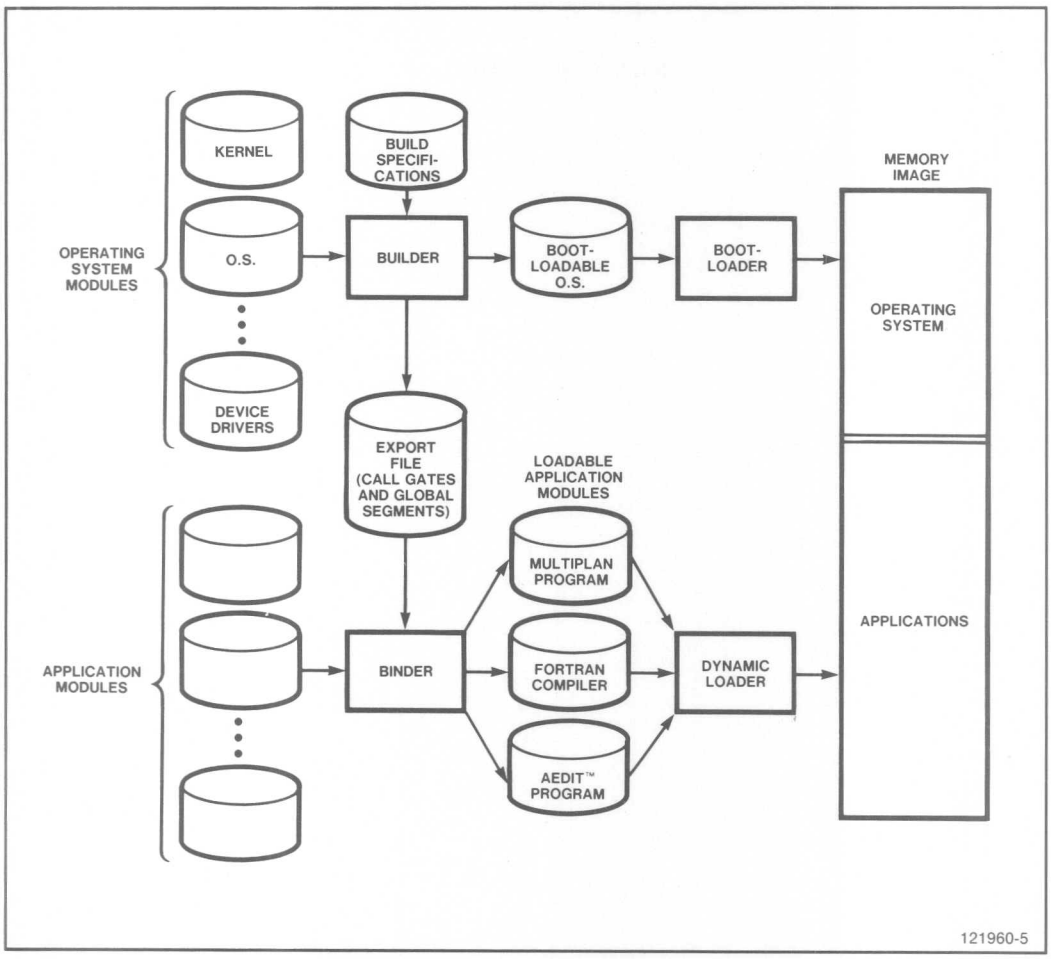


Figure 1-9. Building a Dynamic System

Figure 1-6. Building a Dynamic System

Using Hardware Protection Features

2

CHAPTER 2
USING HARDWARE PROTECTION FEATURES

The architecture of the iAPX 286 enables you to organize software systems so that each task is protected from inadvertent or malicious damage by other tasks and so that privileged procedures are protected from lower-level procedures. You control the degree of protection in your system by the way you set up protection parameters through the Builder or through operating system procedures. The processor interprets the protection parameters and automatically performs all the checking necessary to implement protection.

ADDRESSING MECHANISM

The protection mechanism of the iAPX 286 is embedded in the addressing mechanism. For an introduction to the addressing mechanism, refer to figure 2-1, which shows those portions of the addressing mechanism that are not concerned with protection features. From the point of view of the applications programmer, an address is a pointer. A pointer consists of two parts: a selector and an offset. The

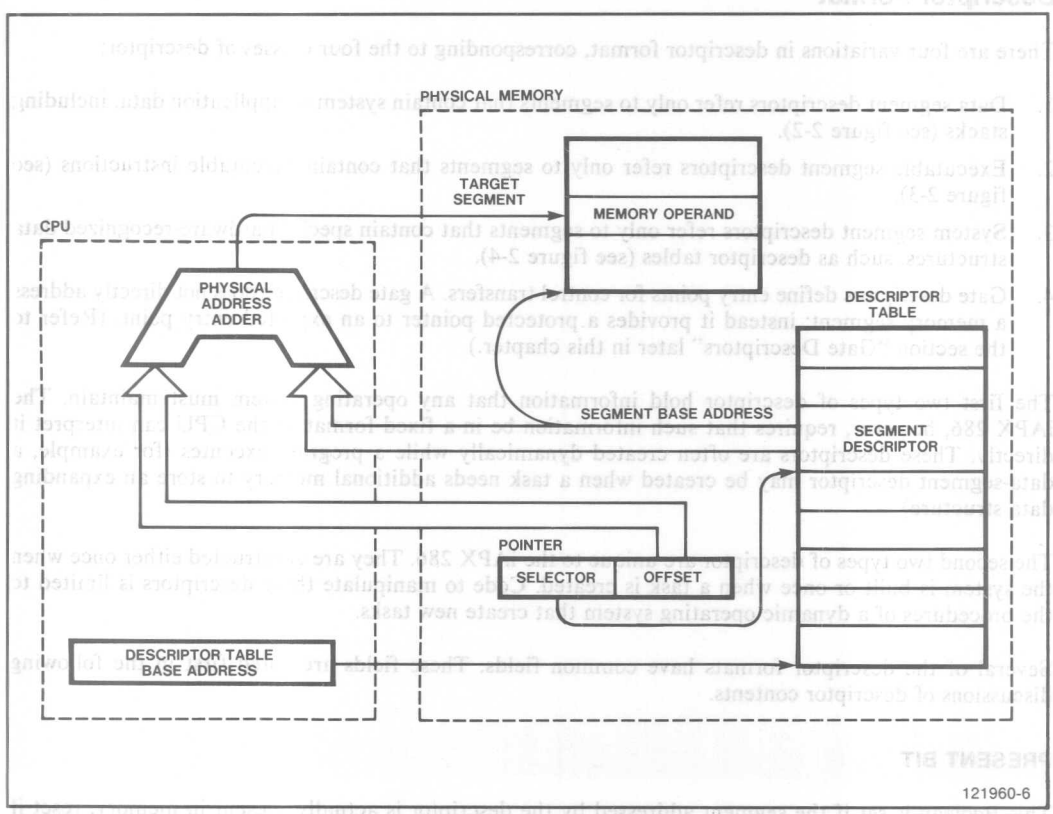


Figure 2-1. Abstraction of Addressing Mechanism

selector portion identifies a segment of the address space and the offset addresses an item within the segment relative to the beginning of the segment.

A selector identifies a segment by reference to a *descriptor table*. Each entry in a descriptor table is a *descriptor*. A selector contains an index that identifies a particular descriptor in a particular descriptor table. The descriptor contains the physical address and access rights of the segment.

Both selectors and descriptors contain additional information that relates to the protection features of the iAPX 286.

DESCRIPTORS

A reference from one segment to another is realized indirectly through a *descriptor*, which contains information about the referenced segment. All descriptors reside in a descriptor table. Every segment must have at least one descriptor; otherwise there is no means to address the segment.

Descriptors are strictly under your control via the Builder or operating system procedures. The existence and function of descriptors is completely invisible to the applications programmer.

Descriptor Format

There are four variations in descriptor format, corresponding to the four classes of descriptor:

1. Data segment descriptors refer only to segments that contain system or application data, including stacks (see figure 2-2).
2. Executable segment descriptors refer only to segments that contain executable instructions (see figure 2-3).
3. System segment descriptors refer only to segments that contain special hardware-recognized data structures, such as descriptor tables (see figure 2-4).
4. Gate descriptors define entry points for control transfers. A gate descriptor does not directly address a memory segment; instead it provides a protected pointer to an exported entry point. (Refer to the section "Gate Descriptors" later in this chapter.)

The first two types of descriptor hold information that any operating system must maintain. The iAPX 286, however, requires that such information be in a fixed format so the CPU can interpret it directly. These descriptors are often created dynamically while a program executes (for example, a data-segment descriptor may be created when a task needs additional memory to store an expanding data structure).

The second two types of descriptor are unique to the iAPX 286. They are constructed either once when the system is built or once when a task is created. Code to manipulate these descriptors is limited to the procedures of a dynamic operating system that create new tasks.

Several of the descriptor formats have common fields. These fields are listed first in the following discussions of descriptor contents.

PRESENT BIT

This Boolean is set if the segment addressed by the descriptor is actually present in memory, reset if not present. Operating systems for dynamic applications that implement virtual memory must set and

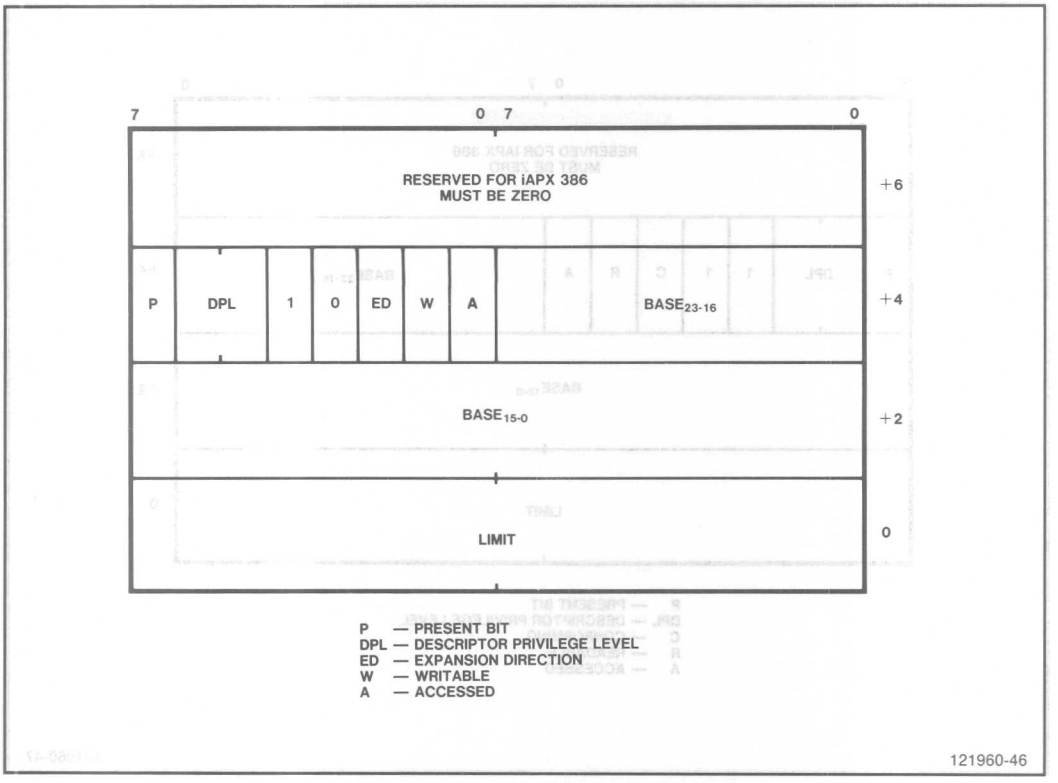


Figure 2-2. Data Segment Descriptor

reset this bit as program segments are brought into or eliminated from memory. Reference to a segment whose present bit is reset causes a fault, providing an opportunity for the operating system to load the segment from virtual store. (Chapter 9 takes up implementation of virtual memory systems.) In systems that do not implement virtual memory, this bit is always set for allocated segments.

DESCRIPTOR PRIVILEGE LEVEL

The value of this item defines the privilege level of the segment addressed by this descriptor. You control the values in the descriptor privilege level (DPL) by the parameters you give to the builder when creating a static system or the resident portion of a dynamic system, or by the procedures your operating system uses when loading segments dynamically.

INTEL RESERVED

This portion of the descriptor is reserved by Intel and should always be initialized with zeros. Other use of this field in a valid descriptor will prevent compatability with the iAPX 386 and other additions to Intel's family of processors.

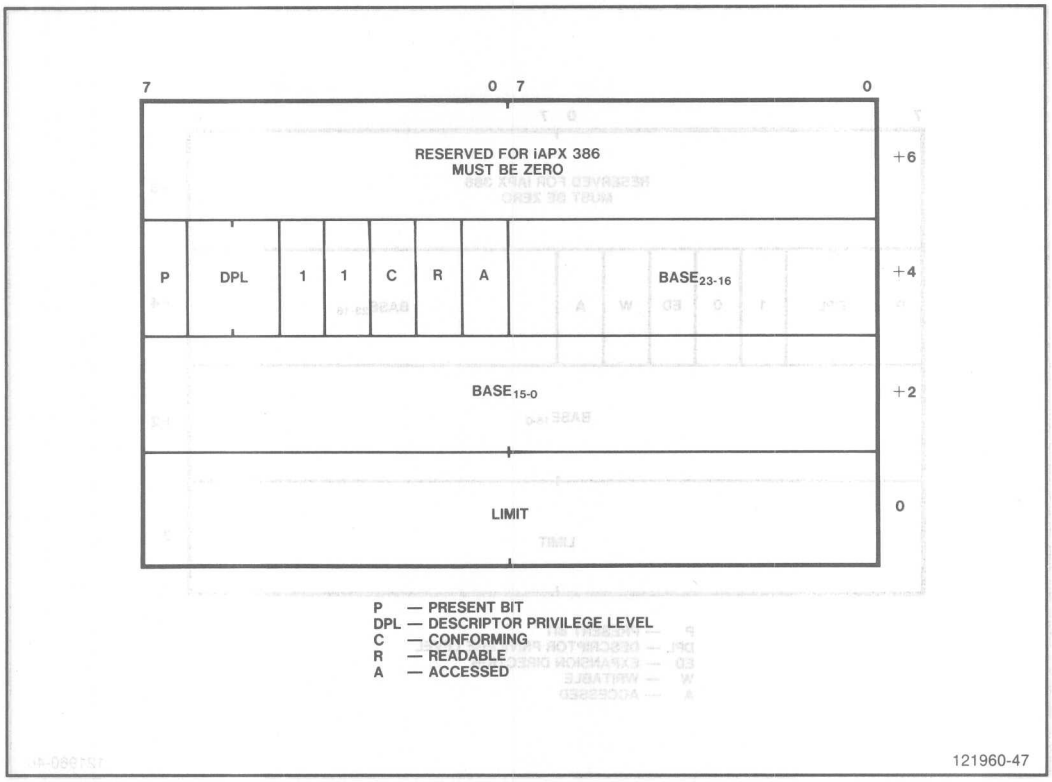


Figure 2-3. Executable Segment Descriptor

SEGMENT BASE

This field contains the physical address of the beginning of the memory segment referred to by this descriptor. The 24 bits of this address give the 80286 a 16-megabyte range of real addresses. This is the only place that physical addresses are used. All other addresses are relative to the physical addresses stored in descriptors, making it possible to relocate executable and data segments without making any changes to the relocated segments or to code that refers to the segments. The only changes necessary to relocate segments are changes to the physical addresses stored in descriptor tables.

You can control the actual location of segments by means of specifications to the Builder or by means of the algorithms your operating system uses to allocate memory to segments that are loaded dynamically.

SEGMENT LIMIT

Segment limits prevent accidental reading or writing beyond the space allocated to a segment. The value of this field is one less than the length of the segment (in bytes) relative to the beginning of the segment. The 16 bits of this field make it possible to have segments up to 64K bytes long. The hardware automatically checks all addressing operations to ensure that they do not exceed the segment limit of the segment to which they refer. This protects other segments from such common programming errors as runaway subscripts.

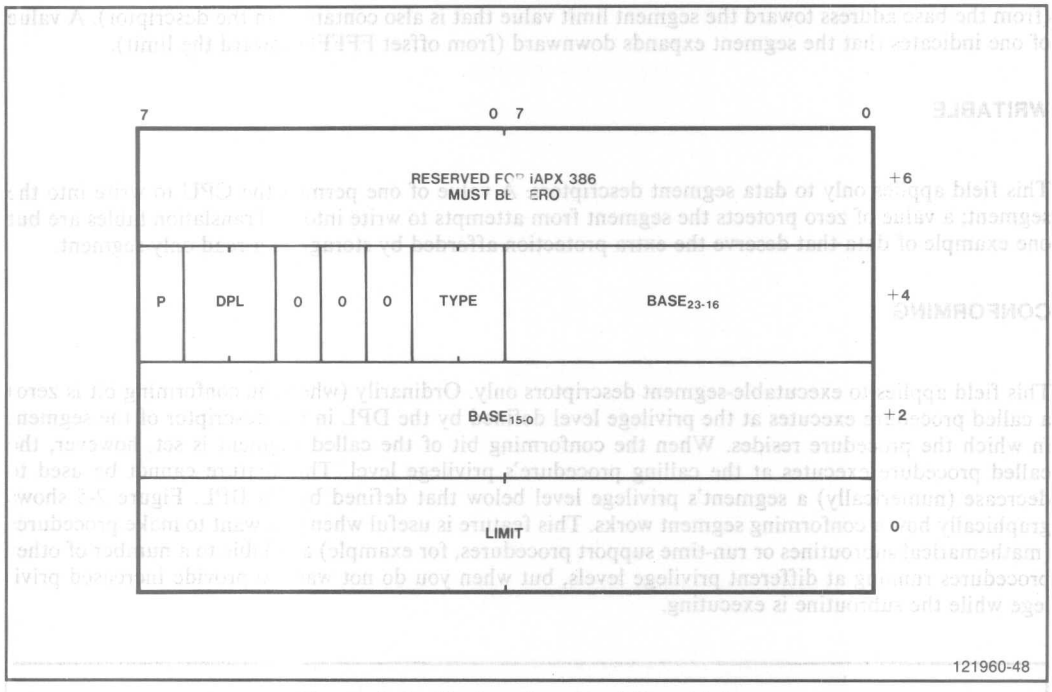


Figure 2-4. System Segment Descriptor

Note that the segment limit field has a different meaning for "expand down" data segments. Refer to the "expansion direction" bit later in this chapter.

SEGMENT TYPE

For system segments, the type field distinguishes between kinds of system segments. System segment types are

1 and 3	Task state segment, a segment used for storing the context of a task. Chapter 4 discusses task state segments more fully.
2	Local descriptor table. The three kinds of descriptor tables are explained later in this chapter.

The processor interprets the type field to ensure that each segment is actually used as intended; for example, an attempt to jump to a local descriptor table is obviously a mistake, and the processor detects this error while examining the target segment's descriptor during the JMP instruction.

EXPANSION DIRECTION

Data segments may contain stacks as well as other data structures. Stacks expand toward lower addresses while most other data structures expand toward greater (higher) addresses. This field indicates the growth pattern for the segment. A value of zero in this field indicates that the segment expands upward

(from the base address toward the segment limit value that is also contained in the descriptor). A value of one indicates that the segment expands downward (from offset FFFFH toward the limit).

WRITABLE

This field applies only to data segment descriptors. A value of one permits the CPU to write into the segment; a value of zero protects the segment from attempts to write into it. Translation tables are but one example of data that deserve the extra protection afforded by storage in a read-only segment.

CONFORMING

This field applies to executable-segment descriptors only. Ordinarily (when the conforming bit is zero) a called procedure executes at the privilege level defined by the DPL in the descriptor of the segment in which the procedure resides. When the conforming bit of the called segment is set, however, the called procedure executes at the calling procedure's privilege level. This feature cannot be used to decrease (numerically) a segment's privilege level below that defined by it's DPL. Figure 2-5 shows graphically how a conforming segment works. This feature is useful when you want to make procedures (mathematical subroutines or run-time support procedures, for example) available to a number of other procedures running at different privilege levels, but when you do not want to provide increased privilege while the subroutine is executing.

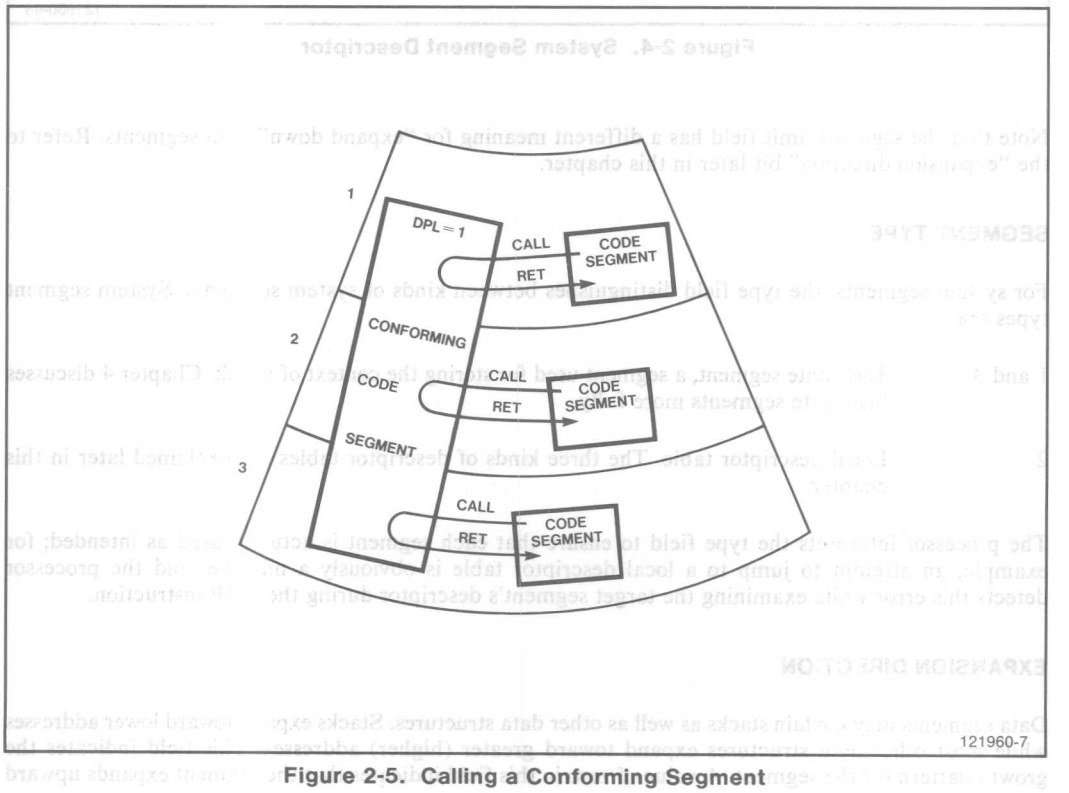


Figure 2-5. Calling a Conforming Segment

READABLE

This field applies only to executable-segment descriptors. When reset, it prevents procedures in other segments from reading this code segment; the contents of the segment can be executed only. It is common, however, for executable segments to contain read-only data, in which case this bit must be set.

ACCESSED

The processor sets this bit when the descriptor is accessed (that is, loaded into a segment register or used by a selector test instruction). Operating systems that implement virtual memory may, by periodically testing and resetting this bit, monitor frequency of segment usage. This bit also indicates whether a segment should be written to secondary storage before the RAM space it occupies is reused.

CONTROL FLOW TRANSFER

Transfers of control are also subject to protection rules. Within the application-oriented part of a task, the protection rules allow unlimited access to code and data. Control transfers to privileged operating-system functions and to other tasks, however, are controlled by gate descriptors. With gate descriptors, the iAPX 286 architecture can perform functions in hardware that operating systems on other processors must do in software. These functions are invoked directly by ordinary CALL and JMP instructions, not by special interrupt or trap instructions.

As table 2-1 illustrates, transfers of control can be classified into four categories, depending on whether control passes to another segment, another privilege level, or another task. This classification can help clarify how privilege levels, descriptor tables, and tasks are used.

Processor functions that cause a change in the flow of control are

- Jump instruction (JMP)
- Procedure call instruction (CALL)
- Procedure return instruction (RET)
- Software interrupt instruction (INT)
- External interrupt

Table 2-1. Categories of Control Flow Transfer

Category	Segment	Privilege Level	Task
Intrasegment	same	same	same
Intersegment	different	same	same
Interlevel	different	different	same
Intertask	same or different	same or different	different

- Interrupt return instruction (IRET)

Control transfers within the same privilege level may be either short (within same segment) or long (to another segment). A short transfer simply specifies the offset of the instruction to which control is transferred in the same segment. A long transfer also uses a selector to identify the segment to which control is transferred.

For control transfer to a different privilege level or different task, the iAPX 286 introduces *gate descriptors.*

Gate Descriptors

A gate descriptor is a type of descriptor used only for transferring control flow to instructions in another segment. Gates provide an indirect reference that is useful for binding and protection purposes. By requiring interlevel and intertask control transfers to reference gate descriptors, the iAPX 286 provides two additional protection features:

1. You can hide a procedure by not providing a gate for its entry point.
2. You can control access to a procedure via the privilege assigned to the gate. This allows hiding critical procedures from untrusted software.

Figure 2-6 illustrates the format of a gate descriptor.

DESTINATION SELECTOR

For call, interrupt, and trap gates, this field contains a selector for the segment descriptor of the destination executable segment. For task gates, the selector in this field points to a descriptor for a task state segment, and the RPL field is not used.

DESTINATION OFFSET

For call, interrupt, and trap gates, this field contains the offset of the entry point within the destination executable segment (not used with task gates).

WORD COUNT

For each privilege level within a task, there is a separate stack. For calls through a call gate, the processor automatically copies parameters from the stack for the calling procedure's privilege level to the stack for the destination's privilege level. In this field, you specify the number of words to copy.

GATE TYPE

For gate descriptors, the type field distinguishes among the four kinds of gates:

0 Call gate
1 Task gate
2 Interrupt gate
3 Trap gate

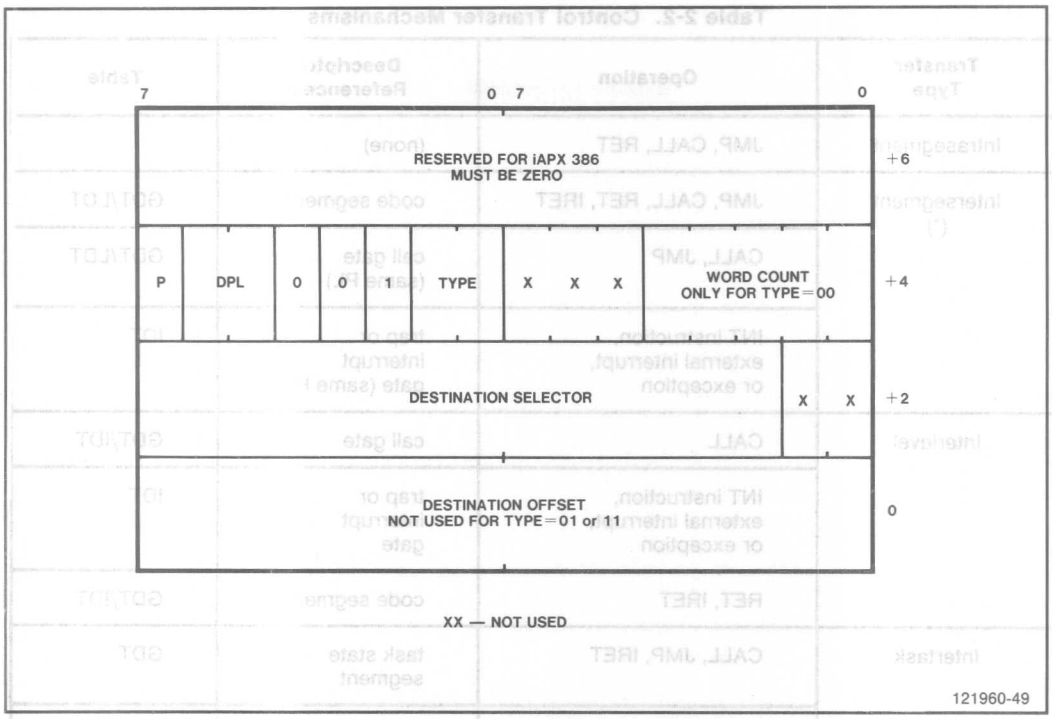


Figure 2-6. Gate Descriptor

PRESENT BIT

Since a gate descriptor does not refer directly to a segment, the present bit in a gate descriptor does not necessarily indicate whether a segment is present. It can be used for other purposes, however. Refer to Chapter 11 for an example of using the present bit to facilitate late binding.

Control Transfer Mechanisms

Table 2-2 summarizes the mechanisms for each class of control flow transfer.

Control transfers within a segment function similarly to intrasegment transfers on the iAPX 86,88, except that the processor checks that the destination address does not exceed the segment limit.

Figure 2-7 illustrates a change in control flow between segments at the *same* privilege level. Any of the following instructions can effect such a transfer:

```
JMP  offset  selector
CALL offset  selector
RET  ;  (offset and selector taken from stack)
```

The *selector* selects a descriptor for an executable segment. The DPL in the target segment's descriptor must be the same as the privilege level under which the calling segment is running. A CALL or

Table 2-2. Control Transfer Mechanisms

Transfer Type	Operation	Descriptor Referenced	Table
Intrasegment	JMP, CALL, RET	(none)	
Intersegment (*)	JMP, CALL, RET, IRET	code segment	GDT/LDT
	CALL, JMP	call gate (same PL)	GDT/LDT
	INT instruction, external interrupt, or exception	trap or interrupt gate (same PL)	IDT
Interlevel	CALL	call gate	GDT/IDT
	INT instruction, external interrupt, or exception	trap or interrupt gate	IDT
	RET, IRET	code segment	GDT/IDT
Intertask	CALL, JMP, IRET	task state segment	GDT
	CALL, JMP	task gate	GDT/LDT
	INT instruction, external interrupt, exception	task gate	IDT

* Includes cases in which the target segment is incidentally the same as the calling segment.

JMP instruction may also reference a call gate. If the target executable segment is at the same privilege level, no level change occurs.

For transfers of control between segments at *different* privilege levels (as illustrated in figure 2-8) there are three differences:

- Only the following instructions can be used:

```
CALL  offset  selector
RET
```

  Jumps between privilege levels within a task are not allowed.

- The *selector* does not select the descriptor of an executable segment but rather selects a *gate descriptor*.
- The *offset* operand must be present but is ignored.

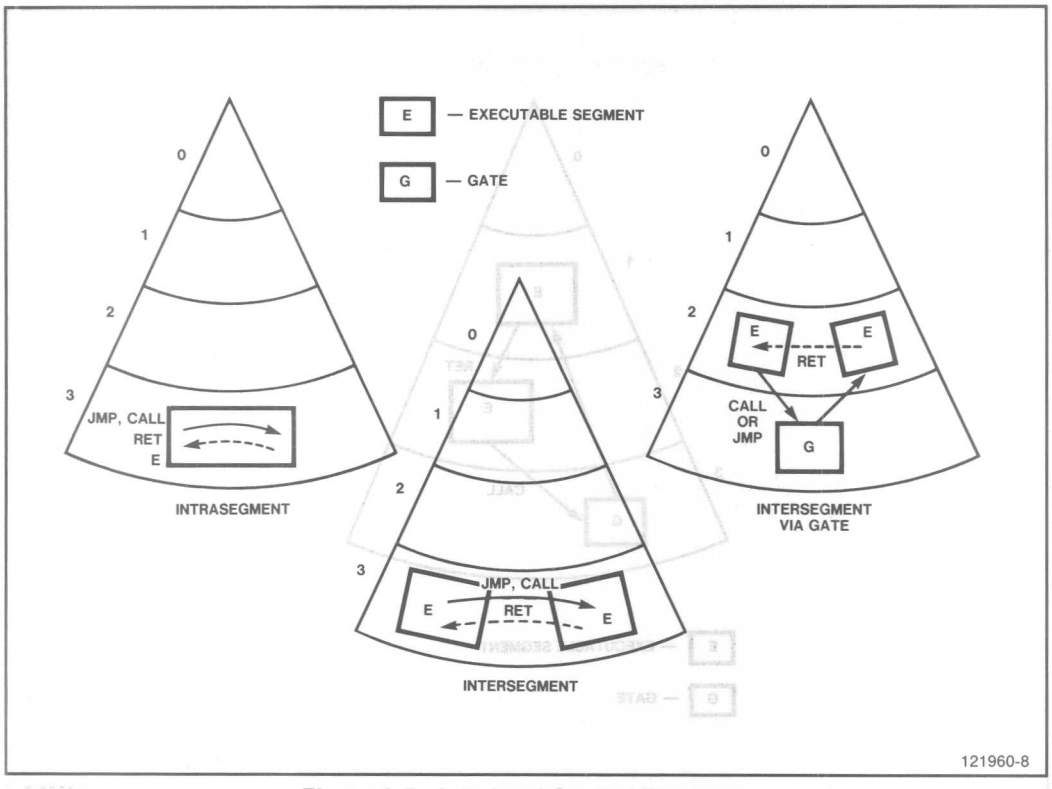


Figure 2-7. Intralevel Control Transfers

Privilege Rules for Gated Intersegment Transfers

An intersegment transfer through a gate involves four privilege level fields:

- The current privilege level (CPL) of the currently executing segment
- The requested privilege level (RPL) in the selector used in the CALL
- The DPL in the gate descriptor
- The DPL in the segment descriptor of the target executable segment

A transfer is valid only if the following relationships among privilege level numbers both hold:

MAX (CPL, RPL) <= gate DPL
target DPL <= CPL

Figure 2-9 illustrates both valid and invalid attempts to perform an interlevel transfer. Path E4,G5,E7 is not valid because the privilege level of gate G5 is numerically less than that of segment E4. Path E4,E8 is not valid because all interlevel transfers must pass through a gate. Path E4,G4,E6 is not valid because the privilege level of E6 is numerically greater than that of G4. Only paths E1,G2,E2; E1,G1,E3; and E2,G1,E3 satisfy the privilege rule above.

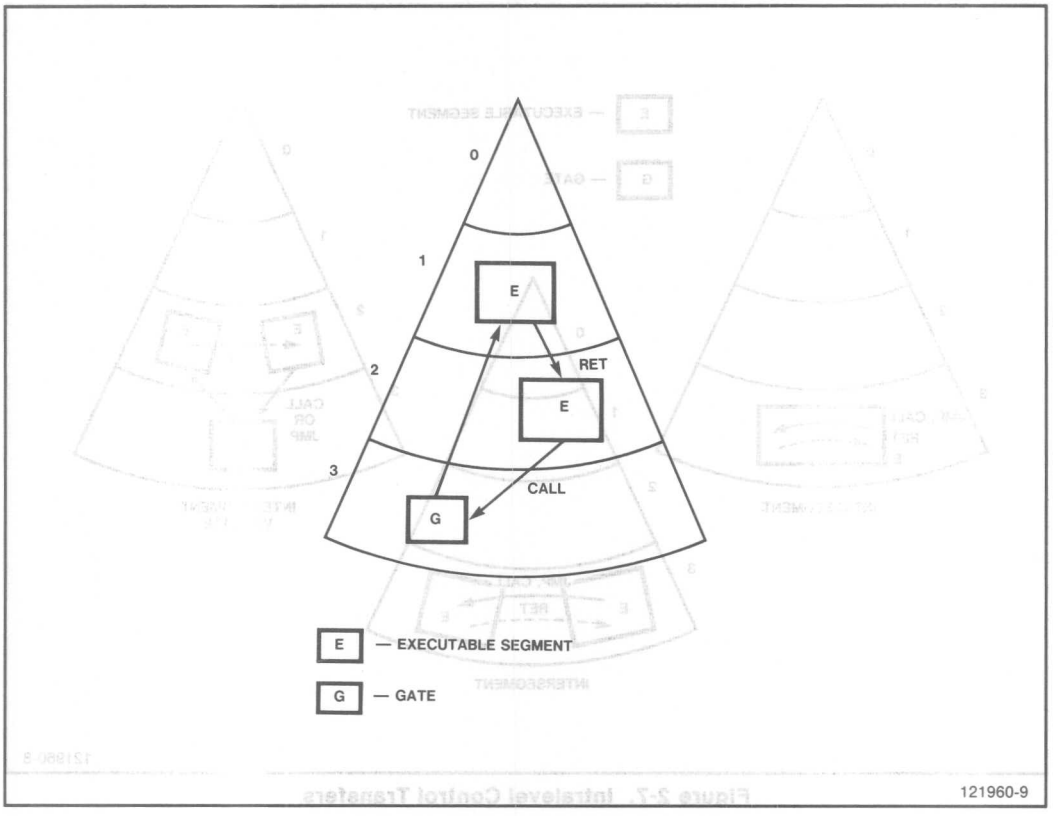


Figure 2-8. Gated Interlevel Call and Return

DESCRIPTOR TABLES

A descriptor table is simply a segment containing an array of eight-byte entries, where each entry is a descriptor. Descriptors are stored in one of three classes of descriptor table:

- Local descriptor table (LDT)
- Global descriptor table (GDT)
- Interrupt descriptor table (IDT)

The descriptors in these tables define *all* the segments in the system. Each table has a variable upper limit, so the size of the table need be no larger than required for the actual number of segments used.

You define the initial contents of descriptor tables through the Builder. An operating system for dynamic applications may change the contents of descriptor tables and may create and delete LDT's as tasks come and go. Correct management of descriptors is the heart of protection on the iAPX 286.

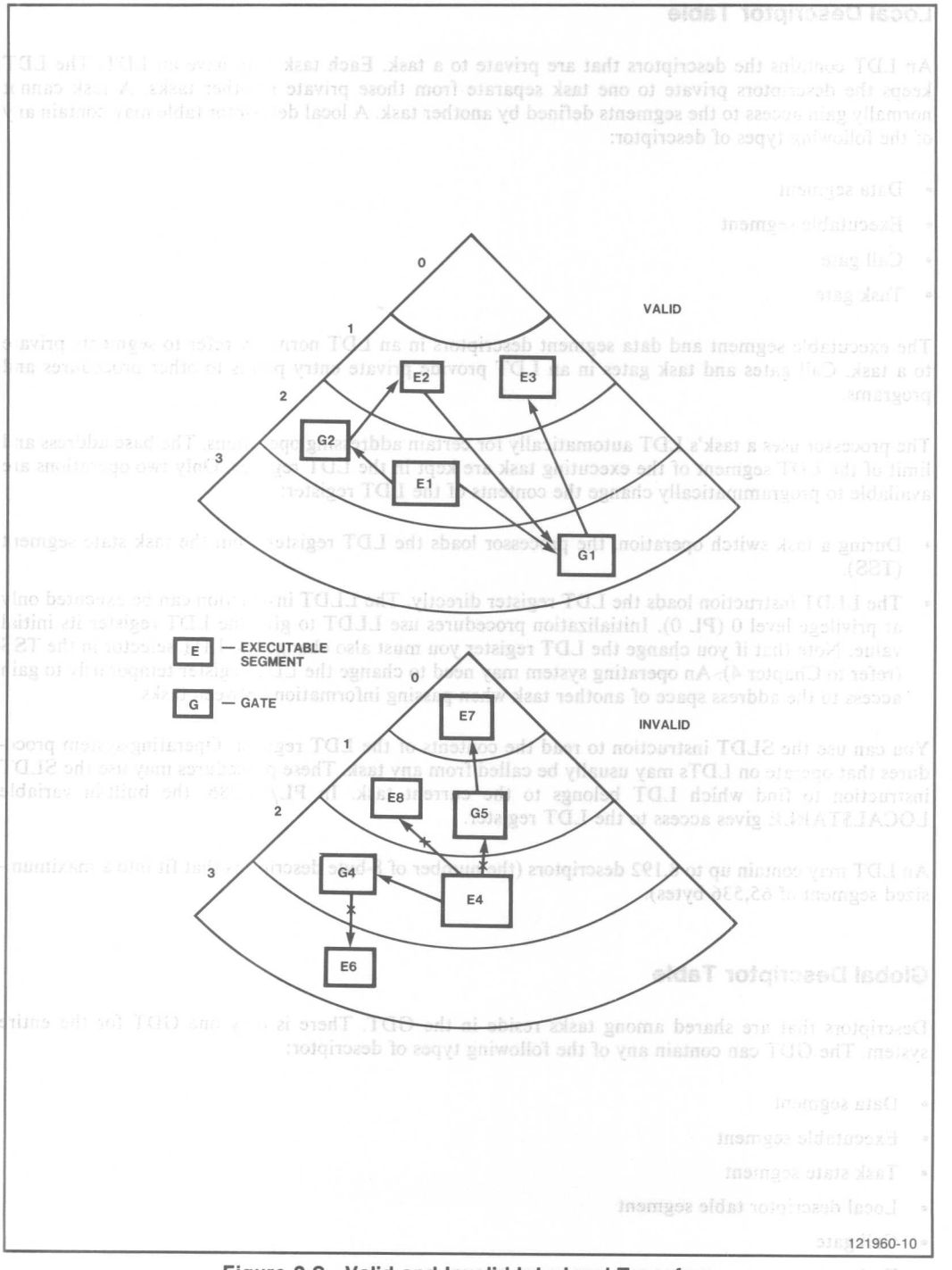


Figure 2-9. Valid and Invalid Interlevel Transfers

Local Descriptor Table

An LDT contains the descriptors that are private to a task. Each task may have an LDT. The LDT keeps the descriptors private to one task separate from those private to other tasks. A task cannot normally gain access to the segments defined by another task. A local descriptor table may contain any of the following types of descriptor:

- Data segment
- Executable segment
- Call gate
- Task gate

The executable segment and data segment descriptors in an LDT normally refer to segments private to a task. Call gates and task gates in an LDT provide private entry points to other procedures and programs.

The processor uses a task's LDT automatically for certain addressing operations. The base address and limit of the LDT segment of the executing task are kept in the LDT register. Only two operations are available to programmatically change the contents of the LDT register:

- During a task switch operation, the processor loads the LDT register from the task state segment (TSS).
- The LLDT instruction loads the LDT register directly. The LLDT instruction can be executed only at privilege level 0 (PL 0). Initialization procedures use LLDT to give the LDT register its initial value. Note that if you change the LDT register you must also change the LDT selector in the TSS (refer to Chapter 4). An operating system may need to change the LDT register temporarily to gain access to the address space of another task when passing information between tasks.

You can use the SLDT instruction to read the contents of the LDT register. Operating-system procedures that operate on LDTs may usually be called from any task. These procedures may use the SLDT instruction to find which LDT belongs to the current task. In PL/M-286, the built-in variable LOCAL$TABLE gives access to the LDT register.

An LDT may contain up to 8,192 descriptors (the number of 8-byte descriptors that fit into a maximum-sized segment of 65,536 bytes).

Global Descriptor Table

Descriptors that are shared among tasks reside in the GDT. There is only one GDT for the entire system. The GDT can contain any of the following types of descriptor:

- Data segment
- Executable segment
- Task state segment
- Local descriptor table segment
- Call gate
- Task gate

Since the GDT is shared among all tasks, its entries are usually protected. The privilege-level field in each descriptor provides this function. When operating-system functions are distributed among and shared by all tasks, the executable segments and data segments of the operating system are normally kept in the GDT. Call gates then provide controlled access to privileged operating system functions.

The processor uses the GDT automatically for certain addressing operations. The base address and limit of the GDT are kept in the processor's GDT register. Only the LGDT instruction (RESTORE$GLOBAL$TABLE in PL/M-286) can alter the contents of the GDT register, and the LGDT instruction can be executed only at PL 0 (i.e., by the operating system).

The SGDT instruction (SAVE$GLOBAL$TABLE in PL/M-286) reads the contents of the GDT register.

A GDT may contain up to 8,191 descriptors (the number of 8-byte descriptors that fit into a maximum-sized segment of 65,536 bytes). The first entry cannot be used as a descriptor. (A null selector is identified by the fact that it refers to this first entry in the GDT.)

Interrupt Descriptor Table

When processing an interrupt, the processor refers to the IDT to determine what interrupt-handling code to execute. Each interrupt is associated with an interrupt identifier, an integer that ranges from 0–255. The interrupt identifier is supplied either by the INT instruction or externally by the processor's INTA cycles. The interrupt identifier indexes an entry in the IDT. An IDT entry may be

- An interrupt gate
- A trap gate
- A task gate

In a manner similar to executable segment and data segment descriptors, each gate descriptor has a descriptor privilege level. The DPL of a descriptor in the IDT determines the privilege required to execute an INT *n* instruction (where *n* is the interrupt indentifier that corresponds to the descriptor). This use of privilege levels prevents unauthorized programs from invoking interrupt handlers.

The processor locates the IDT by way of the IDT register. The IDT register can be changed only by the LIDT (load IDT) instruction (RESTORE$INTERRUPT$TABLE in PL/M-286). Only PL-0 procedures (i.e., the operating system) can execute an LIDT instruction.

The SIDT instruction (SAVE$INTERRUPT$TABLE in PL/M-286) reads the contents of the IDT register.

Refer to Chapters 6 and 7 for more detailed information on how the IDT is used.

SELECTORS

A selector references a segment indirectly by identifying the location in a descriptor table where a descriptor for that segment is stored.

Format of Selector

See figure 2-10 for the format of a selector.

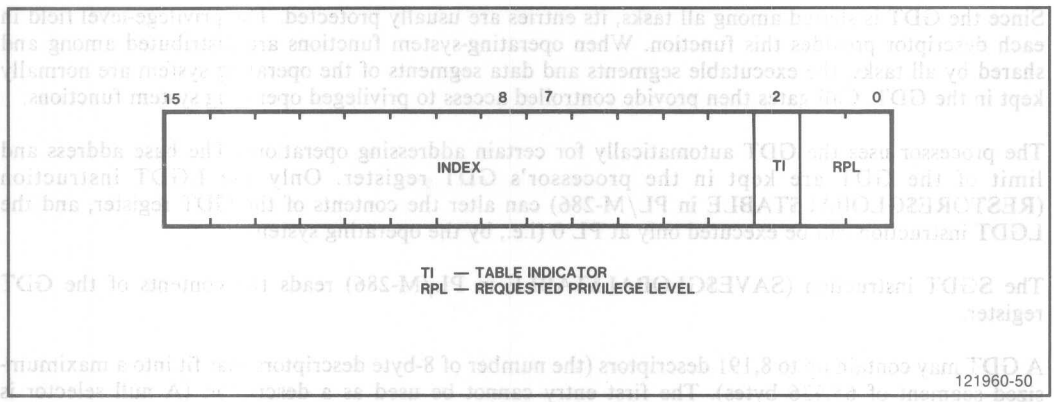


Figure 2-10. Format of a Selector

INDEX

The index field of a selector specifies a descriptor in either the GDT or the task's LDT. The index field may take on values from 0 through $n-1$, where n is the number of descriptors in the table. The processor compares the index with the limit of the descriptor table to ensure that the index refers to a defined descriptor.

TABLE INDICATOR

This bit item tells which descriptor table is indexed by the selector. A value of zero specifies the GDT; one specifies the LDT. (The IDT cannot be referenced via a selector; only via an interrupt identifier.)

REQUESTED PRIVILEGE LEVEL

Selector privilege is specified in the RPL field of a selector. Selector RPL may establish a less trusted privilege level than the current privilege level for the use of that selector. RPL cannot effect an increase in privilege. A task's effective privilege level is the numeric maximum of the RPL and the current privilege level. For example, if a task is executing at PL = 2, an RPL = 3 reduces the task's effective privilege to level 3 for access to that segment. On the other hand, if RPL = 1, the task's effective privilege level remains at 2.

RPL is generally used by an operating system to ensure that selector parameters passed to the more privileged levels of the operating system do not give access to data at a level more privileged than the calling procedure. The RPL field is a convenient place to store the privilege level of the procedure that originated the selector. Any use of the selector can be restricted to the usage allowed its originator.

The ARPL instruction (ADJUST$RPL built-in function in PL/M-286) allows the operating system to set the RPL of a selector either to CPL or to the privilege level of the originator, whichever is (numerically) larger. Refer to Chapter 13 for more information on the use of RPL.

Null Selector

A selector that has a value of zero in both the index and table indicator fields (i.e., appears to point to the first entry of the GDT) is a null selector. You can load such a selector into the DS or ES register, but any attempt to reference memory via that register causes an exception condition.

ALIAS DESCRIPTORS

The need arises in dynamic applications for the operating system to maintain more than one descriptor for a segment; however, care must be taken to preserve system integrity and protection.

As an example of the need for an alternate descriptor, consider the case of an executable segment. Ordinarily, the processor fetches instructions from an executable segment that is typed execute-only. However, if the operating system supports a debugger, the debugger needs to read the executable segment in order to display its contents. The debugger may also need to write to the executable segment in order to set breakpoints. If the debugger tries to use an execute-only segment descriptor to read from or write to the segment, the processor detects a protection exception. To properly use that segment, the debugger must use another descriptor that identifies the segment as a data segment. Figure 2-11 illustrates this situation.

The use of more than one descriptor for a segment is known as *aliasing*. Descriptors used in this way are known as *aliases*, because they provide alternate names for segments.

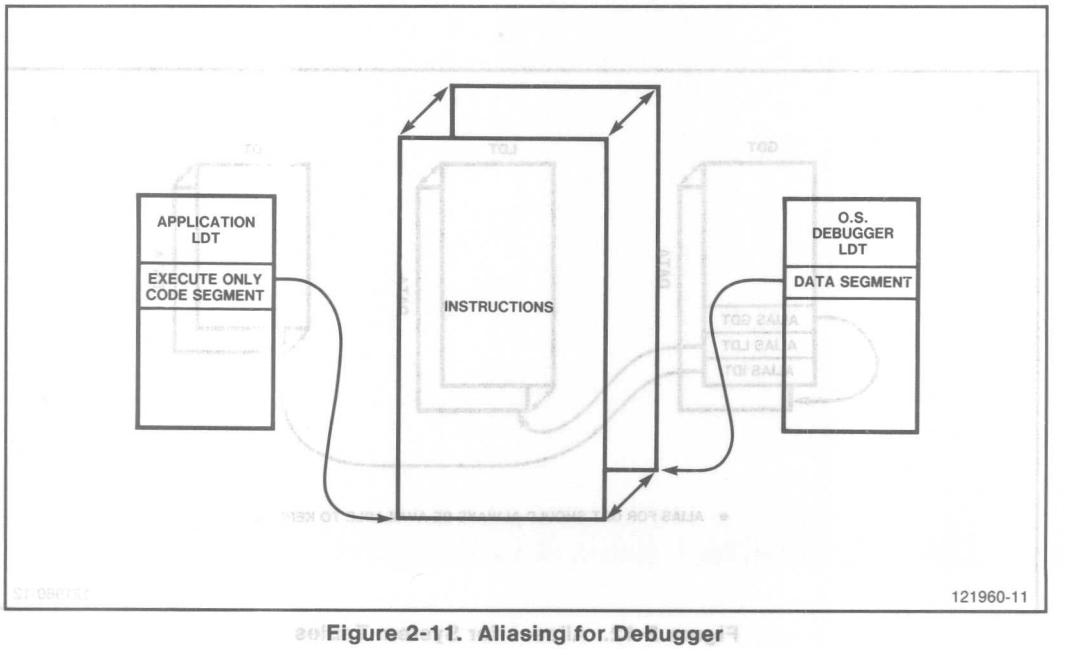


Figure 2-11. Aliasing for Debugger

Explicit Variation of Type

Figure 2-11 illustrates one kind of need for aliasing: the need for a different type specification for a segment. Figure 2-12 shows another example of the same need. In a dynamic application, the operating system may need to modify the GDT, the IDT, TSSs, and LDTs. Changing the interrupt handler for a specific interrupt vector requires changing the IDT. When the operating system places a new segment into the address space of a task (as, for example, when transferring an I/O buffer from an I/O task), it must update the task's LDT. Starting a new task may require modification of the GDT to add descriptors for the new task's LDT and TSS.

With the iAPX 286, however, it is not possible to read or write a system segment by loading its selector into DS or ES. This restriction prevents indiscriminate use of system segments within the operating system. Such use of a system segment requires that the operating system have a descriptor that identifies the system segment as a *data segment*.

The Builder allows for defining aliases for system segments. The Builder, by default, creates data-segment aliases for the GDT and the IDT at fixed locations in the GDT.

Note that aliases for descriptor tables should have PL 0 in order to maintain the integrity of the protection mechanism; otherwise, procedures outside the operating system could indiscriminately change the contents of descriptor tables.

Variation of Length

As illustrated in figure 2-13, aliases for a segment need not always have the same length. In the case shown, the processor's use of a descriptor to a TSS requires only that the segment contain 44 bytes. However, the operating system maintains another descriptor that includes additional information about the task.

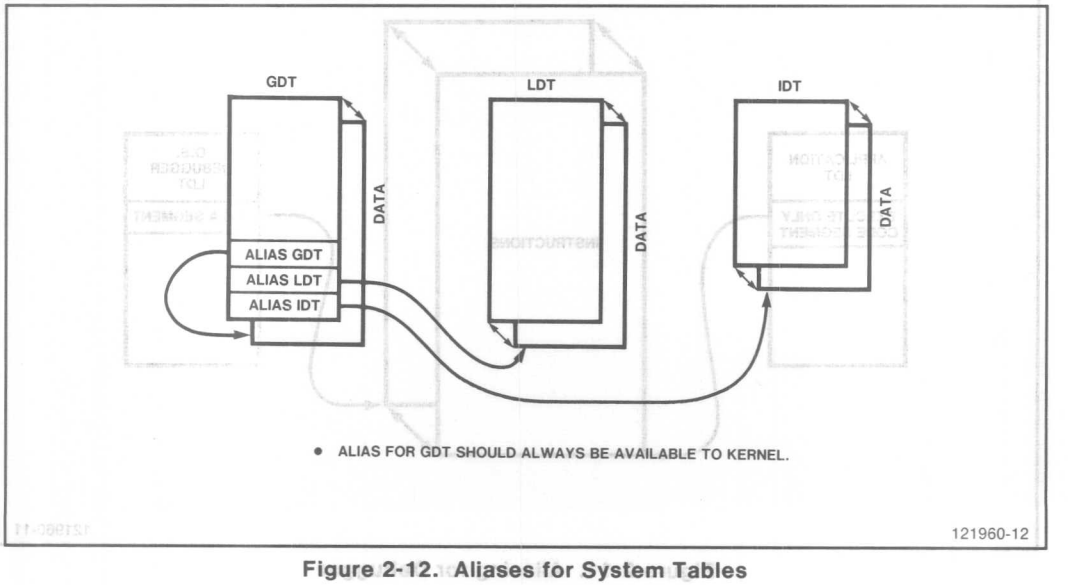


Figure 2-12. Aliases for System Tables

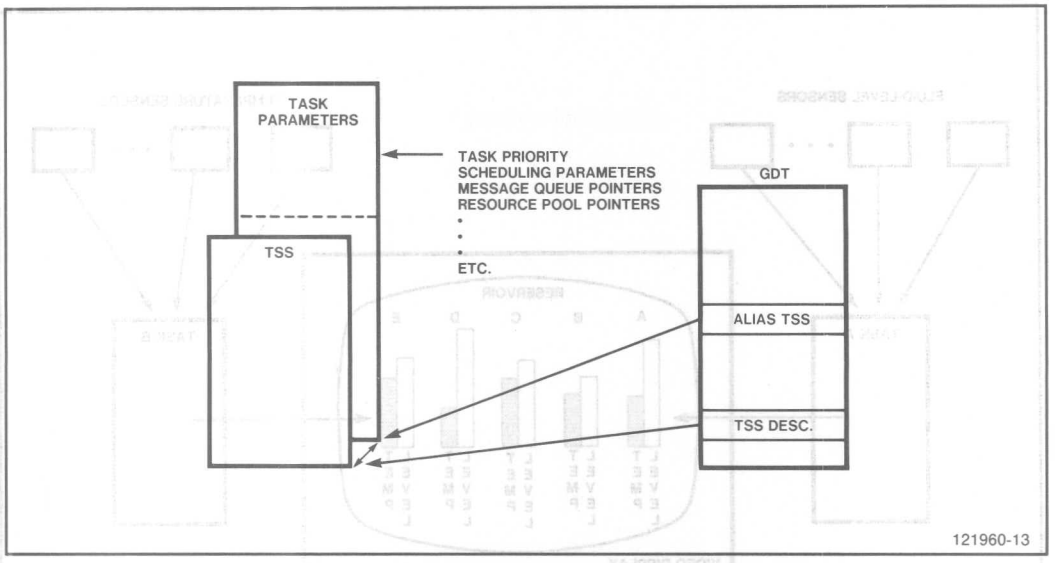


Figure 2-13. Aliases with Differing Limit

Sharing Segments among Tasks

Yet another reason for using aliases is the need for sharing a segment among tasks. Consider an application in which a memory-mapped video display shows status information for a production process. In this application, there are two tasks, each monitoring different aspects of the process but interleaving data on the display (see figure 2-14). Figure 2-15 illustrates how both tasks can access the memory segment containing the display buffer.

You can find segment sharing needs of this sort in both static and dynamic systems. Note that there are other techniques for segment sharing that do not use aliases; for example, placing the segment's descriptor in the GDT, or permitting tasks to share a single LDT. The aliasing technique illustrated here has the advantages that

- No other tasks have access to the display buffer. (Putting its descriptor in the GDT makes it available to all other tasks.)
- Other segments in each of the tasks remain protected from the other task. (With a shared LDT, all segments of each task are accessible from the other.)

Protection and Integrity with Aliasing

You must use aliases with care; improper use can compromise the protection and integrity of your system.

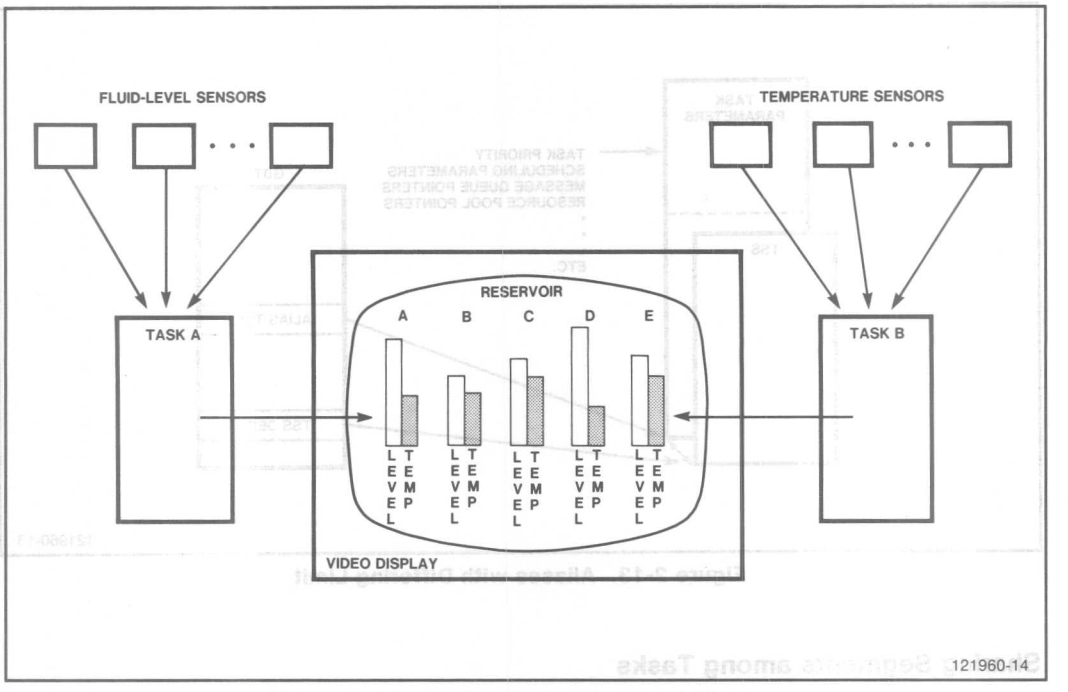


Figure 2-14. Application of Segment Sharing

CONTROL ACCESS TO ALIASES

When you use an alias to provide an alternate type for a system segment (to write to an LDT), any procedure that has access to that alias also has unlimited power to affect the entire system. Therefore, in constructing an operating system that uses such aliases, you must restrict them to the highest privilege levels of the operating system; that is, the DPL of such aliases should always be zero.

PLAN FOR CHANGE

When you design a dynamic system that uses aliases for segment sharing, you must consider what will happen when there is any change in a segment to which aliases are pointing. For example, when a segment is relocated, all descriptors pointing to the relocated segment must be updated. When a segment is deleted, all aliases to it must be nullified. Chapter 5 presents a strategy for handling these changes.

EXAMPLE OF DESCRIPTOR MANIPULATION

As an example of how to manipulate descriptors and descriptor tables using an alias, consider the procedures POINT_AT and NULLIFY in figure 2-16. POINT_AT creates a descriptor at a given slot in the GDT. NULLIFY invalidates a descriptor in the GDT. It is intended for use in connection with POINT_AT to prevent accidental use of descriptors that are no longer needed.

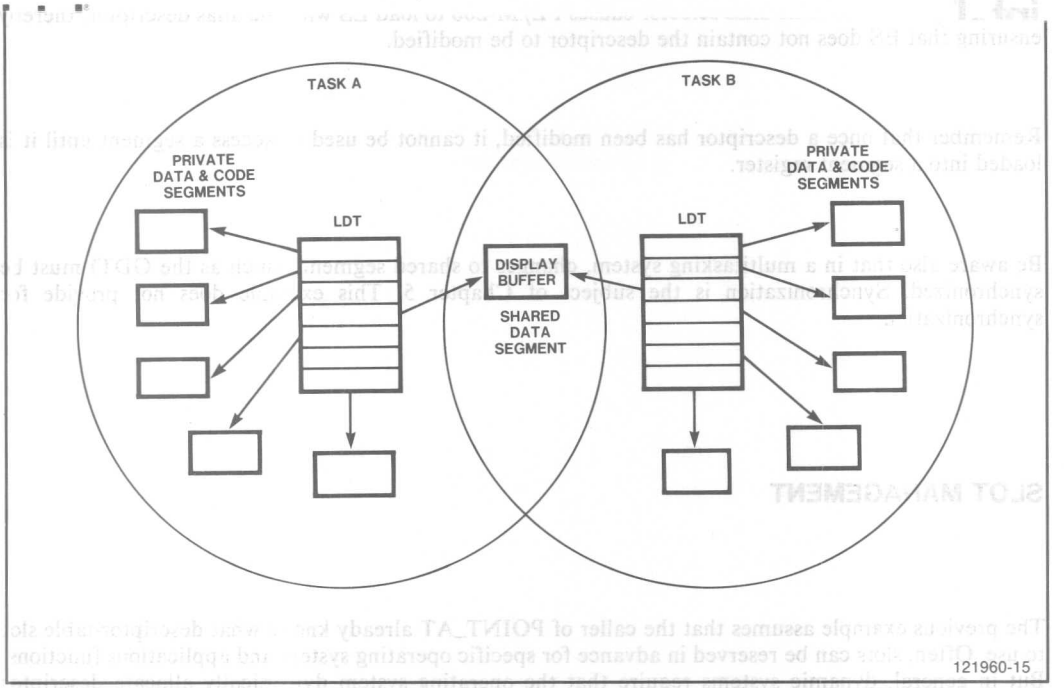


Figure 2-15. Aliases for Segment Sharing

NULLIFY invalidates a descriptor by writing a value of 80H in the access rights byte. A value of 80H is invalid because it indicates a system segment of type zero, but no type zero is defined for system segments.

POINT_AT purposely loads the access rights byte of the descriptor last, to ensure that an accidental use of the descriptor (as might occur if an external interrupt gives control to another procedure or task) does not find partially complete information in a descriptor that otherwise looks valid.

These procedures do no checking of the privilege level of the calling procedure, and they freely create descriptors of any type (except gate descriptors) and any DPL. Therefore, they are suitable for use only at PL 0. As long as no gates for these procedures are provided at another privilege level, they can be called only by other PL-0 procedures. For an example of how you might use such procedures, refer to the example of a memory manager in Chapter 3.

When writing code to manipulate descriptors, you must be careful about changing a descriptor that is currently loaded in either of the processor's data-segment registers (DS or ES). Either disable interrupts or move zero to the register before changing the descriptor, for example:

```
MOV  DS,0
```

Failure to do this leaves open the possibility that an external interrupt may cause the processor to reload the segment register using the partially modified descriptor. When coding in PL/M-286, use the compiler's CODE option to view the way the generated code handles the DS and ES registers. This situation does not arise in the present example. DS points to the data segment that contains the sole

global data item GDTA_SEL. PL/M-286 uses ES for all the BASED variables used here. Referencing the descriptor table via its alias selector causes PL/M-286 to load ES with the alias descriptor, thereby ensuring that ES does not contain the descriptor to be modified.

Remember that once a descriptor has been modified, it cannot be used to access a segment until it is loaded into a segment register.

Be aware also that in a multitasking system, changes to shared segments (such as the GDT) must be synchronized. Synchronization is the subject of Chapter 5. This example does not provide for synchronization.

SLOT MANAGEMENT

The previous example assumes that the caller of POINT_AT already knows what descriptor-table slot to use. Often, slots can be reserved in advance for specific operating system and applications functions. But in general, dynamic systems require that the operating system dynamically allocate descriptor table slots.

Figure 2-17 illustrates a way of identifying available slots in a descriptor table. A value of zero in the access rights byte is invalid for any descriptor, so it can mark a free slot. A value of 80H (as used in the previous example) is also invalid and can mark a reserved but unused slot.

In larger systems, the time needed to search a descriptor table linearly for free slots may become excessive. Shorter search times may result from linking available slots together in a manner similar to that shown in figure 2-17. Contiguous free slots are treated as a block, with a count of the number of slots in the first and last slots of the block. (Note that the block size is stored in the reserved word of available descriptor slots. Since available descriptor slots contain an invalid type code, this use of the reserved word does not prevent upward compatibility.) All the free blocks are linked in a circular, two-way list that includes the list header. The list header can reside at a fixed slot location that is the same in all descriptor tables (in the case of figure 2-17 the list header is at slot number one). Algorithms normally used for managing memory space may also apply to blocks of free descriptors. Refer to Chapter 3 for an example of such an algorithm.

It is convenient for the operating system to use adjacent descriptor slots for related purposes; for example, by locating together all the descriptors that the operating system uses for one task, the operating system can quickly find any of those descriptors as long as it knows the location of one. Therefore, the algorithm used for slot management should combine adjacent free slots into a single block. The procedures used to manage free slots should then have a parameter that specifies the number of adjacent slots.

```
PL/M-286 COMPILER     960-505                    date                PAGE   1

system-ID  PL/M-286 Vx.y COMPILATION OF MODULE POINT
OBJECT MODULE PLACED IN :F1:POINT.OBJ
COMPILER INVOKED BY:  PLM286.86 :F1:POINT.PLM


                $ PAGEWIDTH(71) TITLE('960-505') INCLUDE (:F1:NUCSUB.PLM)
          =     $ NOLIST

   1            POINT:  DO;
                /*********************************************************/
                /*              Global declarations.                     */
   2    1       DECLARE DESC_STR LITERALLY
                 'LIMIT      WORD,
                  LO_BASE    WORD,     /* Format of a descriptor. */
                  HI_BASE    BYTE,
                  RIGHTS     BYTE,
                  SW_RESRVD WORD';
   3    1       DECLARE DT_SIZE    LITERALLY '200',
                        GDTA_SEL   SELECTOR, /* Points to GDT alias */
                        GDTA_WSEL WORD AT (@GDTA_SEL)
                          INITIAL (8), /* Slot #1 by convention */
                        GDT        BASED GDTA_SEL (DT_SIZE)
                                   STRUCTURE (DESC_STR);
                /*********************************************************/
                /*  Subroutine to determine either the alias for         */
                /*  the GDT or the alias for this task's LDT,            */
                /*  depending on the TI bit in SEL.                      */

   4    1       FIND_DT_ALIAS: PROCEDURE (SEL) SELECTOR
                               PUBLIC REENTRANT;

   5    2        DECLARE SEL       SELECTOR,
                        WSEL      WORD AT (@SEL);
   6    2        DECLARE LDT_SEL   SELECTOR,
                        LDT_WSEL  WORD AT (@LDT_SEL);

   7    2        IF (WSEL AND 0004H)=0
                  THEN /* It's a selector to the GDT. */
   8    2          RETURN GDTA_SEL;

   9    2          ELSE DO; /* It's a selector to this task's LDT. */
  10    3            LDT_SEL=LOCAL$TABLE; /* PL/M 286 built-in; stores a
                     selector to the GDT descriptor for this task's LDT. */
  11    3            LDT_WSEL=LDT_WSEL+8; /* Add 1 to index field. */
                       /* By convention, next slot holds alias. */
  12    3            RETURN LDT_SEL;
  13    3          END;

  14    2       END FIND_DT_ALIAS;
                $ EJECT
```

Figure 2-16. Descriptor Manipulation Example

```
                /*******************************************************/
                /*  Create a descriptor at a given slot to a given     */
                /*  segment, and return selector that points           */
                /*  to that slot.                                      */

 15   1         POINT_AT: PROCEDURE(SLOT, RIGHTS, PHYS_ADDR_PTR, LIMIT)
                         PUBLIC REENTRANT;

 16   2          DECLARE SLOT        SELECTOR,
                        SLOTW       WORD AT (@SLOT), /* Alternate type */
                        RIGHTS      BYTE,
                        PHYS_ADDR_PTR POINTER,
                        PHYS_ADDR   BASED PHYS_ADDR_PTR STRUCTURE
                         (LO_WORD WORD,
                          HI_WORD WORD),
                        LIMIT           WORD;

 17   2          DECLARE SLOTI     WORD,     /* Slot index */
                        DTA_SEL   SELECTOR, /* To be set to either
                                               GDT alias or LDT alias. */
                        DT        BASED DTA_SEL  (DT_SIZE)
                                  STRUCTURE (DESC_STR);

 18   2          DTA_SEL = FIND_DT_ALIAS(SLOT);
 19   2          SLOTI = SHR(SLOTW,3); /* Expose index value. */
 20   2          DT(SLOTI).LO_BASE   = PHYS_ADDR.LO_WORD;
 21   2          DT(SLOTI).HI_BASE   = LOW(PHYS_ADDR.HI_WORD);
 22   2          DT(SLOTI).LIMIT     = LIMIT;
 23   2          DT(SLOTI).SW_RESRVD = 0;
 24   2          DT(SLOTI).RIGHTS    = RIGHTS;
 25   2          RETURN;

 26   2         END POINT_AT;

                /*******************************************************/
                /*        Invalidate descriptor indexed by SLOT.       */

 27   1         NULLIFY: PROCEDURE (SLOT) PUBLIC REENTRANT;

 28   2          DECLARE SLOT     SELECTOR,
                        SLOTW    WORD AT (@SLOT); /* Alternate type */

 29   2          DECLARE SLOTI     WORD,     /* Slot index */
                        DTA_SEL   SELECTOR, /* To be set to either
                                               GDT alias or LDT alias. */
                        DT        BASED DTA_SEL  (DT_SIZE)
                                  STRUCTURE (DESC_STR);

 30   2          DTA_SEL = FIND_DT_ALIAS(SLOT);
 31   2          SLOTI = SHR(SLOTW,3); /* Get index part of selector. */
 32   2          DT(SLOTI).RIGHTS = 80H;/* This invalid value prevents
                                        use of the descriptor.      */
 33   2          RETURN;

 34   2         END NULLIFY;

                /*******************************************************/
```

Figure 2-16. Descriptor Manipulation Example (Cont'd.)

```
PL/M-286 COMPILER     960-505                 date                  PAGE   3

  35   1       END POINT;


MODULE INFORMATION:

     CODE AREA SIZE      = 00CAH     202D
     CONSTANT AREA SIZE  = 0000H       0D
     VARIABLE AREA SIZE  = 0002H       2D
     MAXIMUM STACK SIZE  = 0018H      24D
     129 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     5KB MEMORY USED    (5%)
     0KB DISK SPACE USED

END OF PL/M-286 COMPILATION
```

Figure 2-16. Descriptor Manipulation Example (Cont'd.)

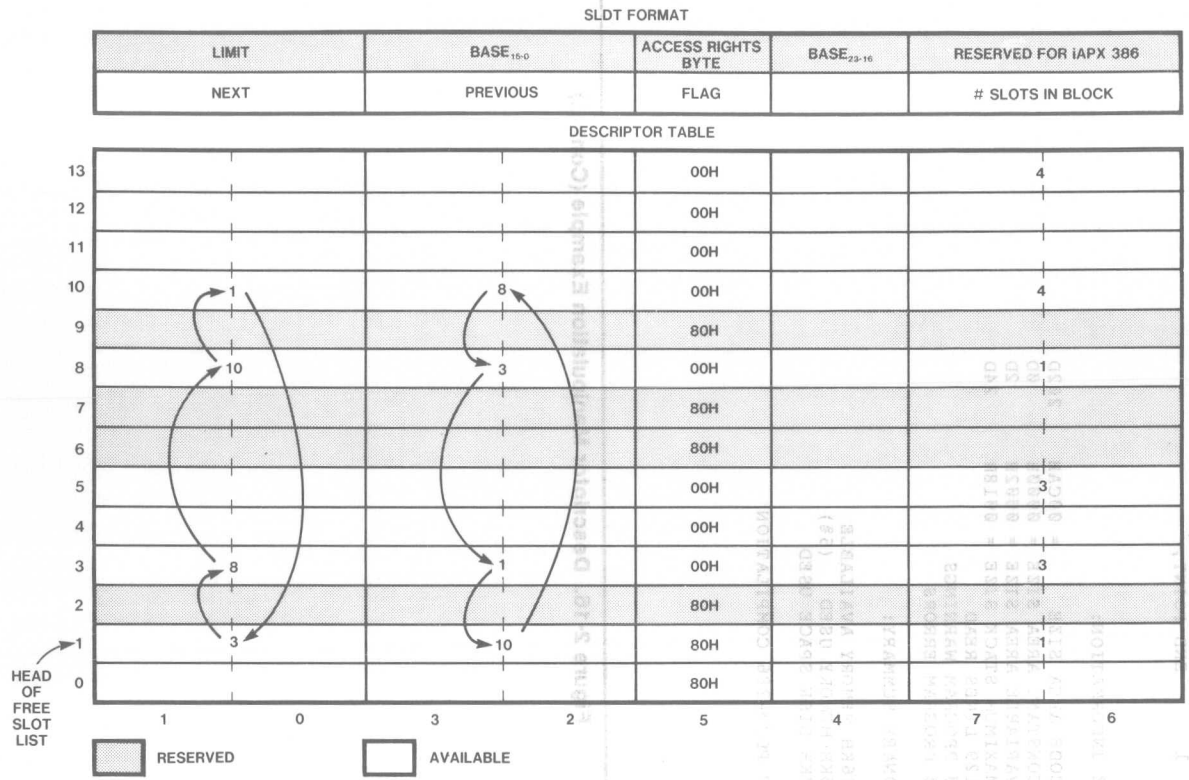


Figure 2-17. Available Slot List

121960-51

121960-001

Real Memory Management 3

3 Real Memory Management

CHAPTER 3
REAL MEMORY MANAGEMENT

In dynamic applications, when tasks begin and end frequently, the operating system is responsible for allocating memory to tasks. Without the control that an operating system provides, independent tasks cannot be trusted to share the system's memory harmoniously. The iAPX 286, through the descriptor mechanism, gives the operating system the power to control memory usage. For static systems, you can use the Builder to allocate memory. This chapter presents an example of how to manage real memory dynamically, using dynamically created descriptors.

MEMORY MANAGEMENT FUNCTIONS

Procedures at various levels in a system have a need to get memory for their use. Some of the functions that use memory dynamically include

- Loading the code and data segments for a new task
- Creating the TSS and LDT of a new task
- Expanding an application data structure
- Expanding system stack segments when stacks grow too large
- Allocating buffers for a newly opened file

In allocating memory statically using the Builder, you or the Builder must keep account of what memory locations are available and what locations are used. A dynamic memory allocation module must do the same, but, in addition, it needs to reuse the space vacated by tasks that have finished. Even tasks that are still executing may no longer need all of the memory they once were using, so you need to provide some means for them to return that space dynamically. The need to reclaim formerly used memory space provides considerable challenge to operating system designers.

Protection usually requires that segments not overlap. The operating system should accurately keep track of allocated memory to prevent new segments from overlapping current segments.

Allocation of memory to tasks in a dynamic environment is complicated by the facts that segments have differing lengths and that the order of creation and deletion is unpredictable. Consequently, after a number of tasks have come and gone, memory becomes fragmented, as illustrated in figure 3-1. It becomes increasingly difficult to find free areas large enough to accomodate requests for space. It may happen that no single free area in all of memory is large enough to fill a request for memory even though the total of all smaller available areas is larger than the amount needed. Knuth (see "External Literature" in the Preface) discusses how various memory-management mechanisms can minimize or magnify this problem.

Memory management on the iAPX 286 differs from memory management on other processors in that descriptors must be constructed to access any region of physical memory.

EXAMPLE OF A MEMORY MANAGER

As an example of how a memory manager can manipulate descriptors and segments, consider a memory management module that implements a version of the "first fit" algorithm (as described by Knuth) and combines adjacent free segments as a way to reduce fragmentation. This example employs the

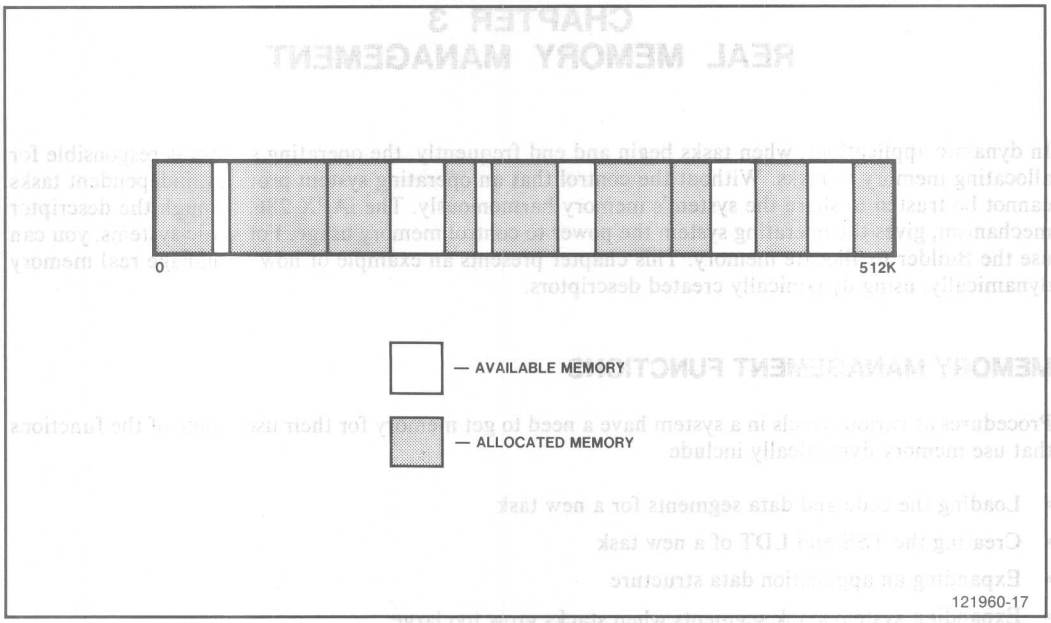


Figure 3-1. Memory Fragmentation

"first fit" algorithm because it provides an opportunity to illustrate how to create descriptors dynamically to access free memory areas, not because it necessarily performs best in any specific application. Knuth discusses other algorithms, including the "buddy system."

Figure 3-2 illustrates conceptually the structure of this module. Hidden inside the module are the list of available memory space, the aliases that permit modification of the GDT and LDTs, and the space-management algorithms. The PUBLIC procedures ALLOCATE and FREE are the only interfaces with the world outside the module.

Data Structures

Figure 3-3 illustrates the data structures implemented in this example memory manager. One-word tags bound every memory area on the low and the high end. These *boundary tags* indicate whether the area is free or in use by some task. Free areas are chained into a two-way linked list that uses physical addresses to point to the next and prior segments in the list. The size of an area (in bytes) is stored with the link addresses in the low-addressed end of the area. At the high-addressed end, a physical address points to the beginning of the area.

With boundary tags at both ends of every memory area, the memory manager can, when freeing a segment that is no longer needed, easily determine whether either of the adjoining areas is free. Figure 3-4 illustrates how adjacent free areas can be combined to reduce memory fragmentation.

Figure 3-5 illustrates that boundary tags are invisible to procedures outside the memory management module because the tags are not within the base and limit addresses in the segment descriptor that the memory manager returns to the caller.

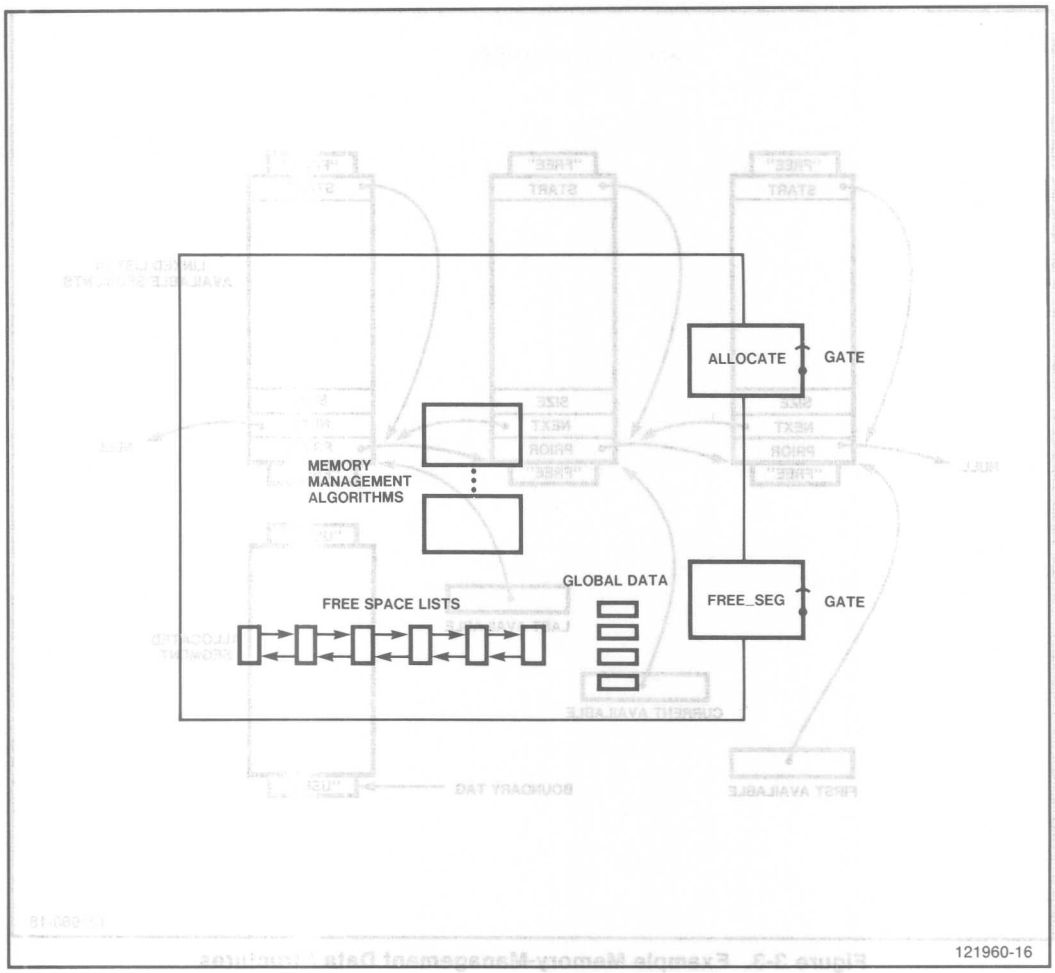


Figure 3-2. Information Hiding in Memory-Management Example

This memory manager does not maintain descriptors to free segments; instead, it creates descriptors dynamically when it needs to address the boundary tags and space-management linkages. This policy minimizes the number of descriptors in the GDT. Note that as a result, free areas may be larger than 64K bytes.

Physical addresses are stored as double words (DWORD) to take advantage of double word arithmetic in PL/M-286. In an actual implementation, you may wish to store physical addresses in three-byte fields to save space, but you would need to implement arithmetic operations for three-byte operands.

The total size of the memory-management items associated with each free area is 20 bytes; therefore, to ensure that no memory area is too small to contain the memory-management items, this algorithm chooses the size of the allocated space to be a multiple of the next integer greater than 20 that is also a multiple of 16. This number is identified as BLOCK_MODULUS.

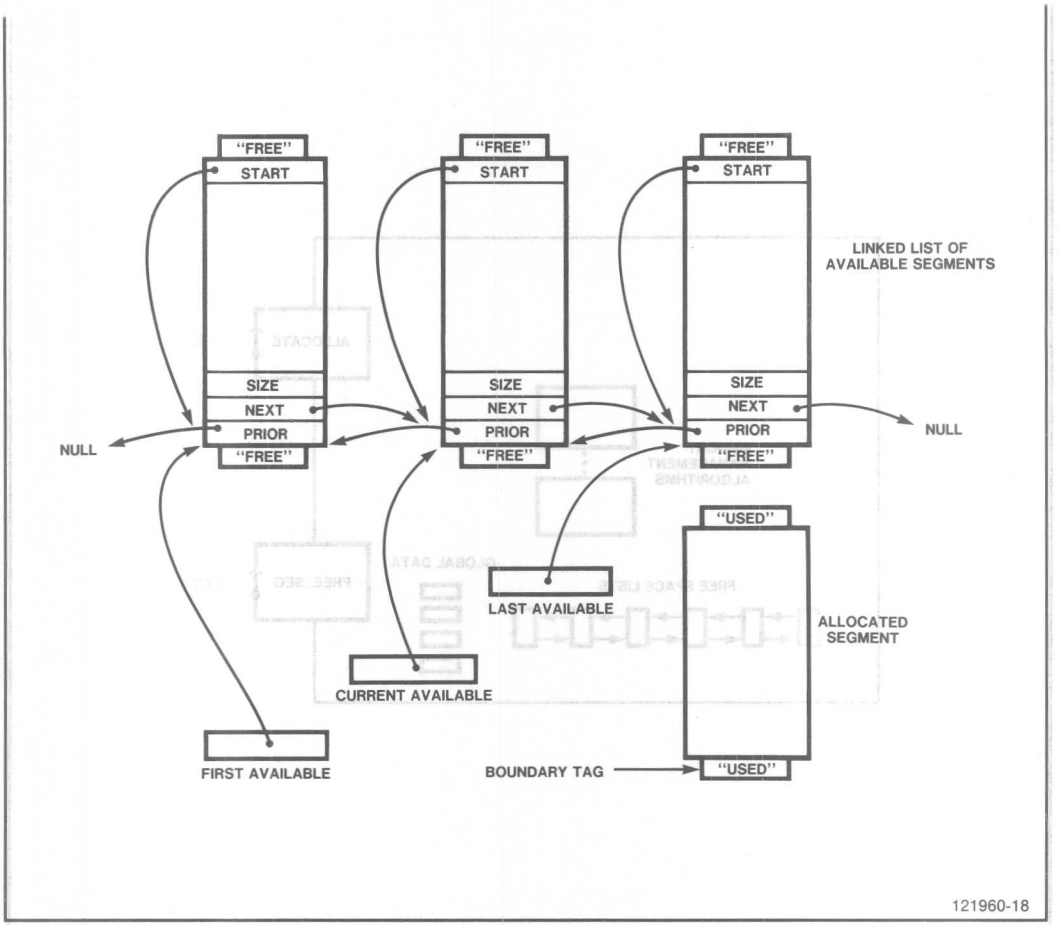


Figure 3-3. Example Memory-Management Data Structures

The memory manager maintains physical-address pointers to the first segment in the available segment list and to the last. It also maintains a "current available" pointer that corresponds to Knuth's "roving pointer."

This example also makes some assumptions about the placement of descriptors in the GDT. Figure 3-6 illustrates these assumptions:

- The memory manager frequently needs to create temporary descriptors for such purposes as reading and updating the boundary tags and link fields. For its own convenience, the memory manager reserves GDT slots for these temporary descriptors. This example identifies the slots as SLOT_A, SLOT_B, and SLOT_C.
- Adjacent slots in the GDT contain all the descriptors to information for one task. This way, given a selector for one task descriptor, simple addition or subtraction yields selectors for other descriptors for the same task.

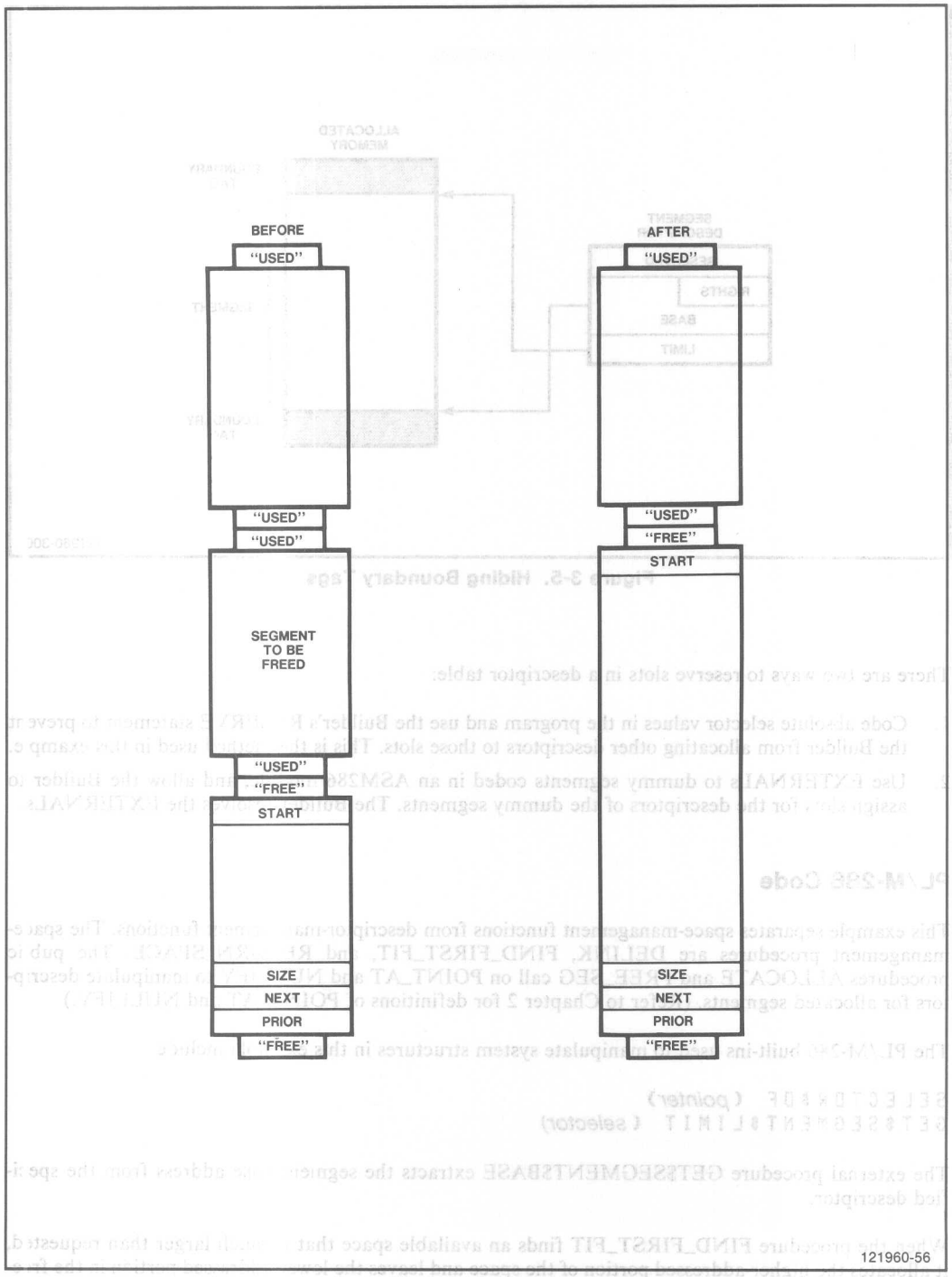


Figure 3-4. Using Boundary Tags

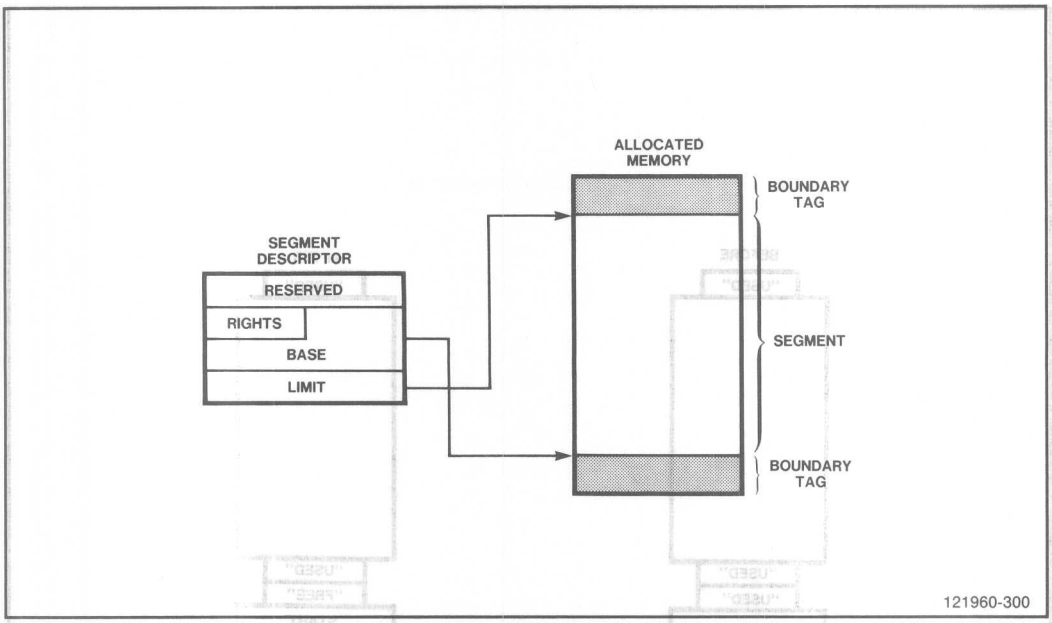


Figure 3-5. Hiding Boundary Tags

There are two ways to reserve slots in a descriptor table:

1. Code absolute selector values in the program and use the Builder's RESERVE statement to prevent the Builder from allocating other descriptors to those slots. This is the method used in this example.
2. Use EXTERNALs to dummy segments coded in an ASM286 module, and allow the Builder to assign slots for the descriptors of the dummy segments. The Builder resolves the EXTERNALs.

PL/M-286 Code

This example separates space-management functions from descriptor-management functions. The space-management procedures are DELINK, FIND_FIRST_FIT, and RETURN_SPACE. The public procedures ALLOCATE and FREE_SEG call on POINT_AT and NULLIFY to manipulate descriptors for allocated segments. (Refer to Chapter 2 for definitions of POINT_AT and NULLIFY.)

The PL/M-286 built-ins used to manipulate system structures in this example include

```
SELECTOR$OF  (pointer)
GET$SEGMENT$LIMIT  (selector)
```

The external procedure GET$SEGMENT$BASE extracts the segment base address from the specified descriptor.

When the procedure FIND_FIRST_FIT finds an available space that is much larger than requested, it allocates the higher-addressed portion of the space and leaves the lower-addressed portion in the free-space list. Figure 3-7 illustrates the process of splitting an available space.

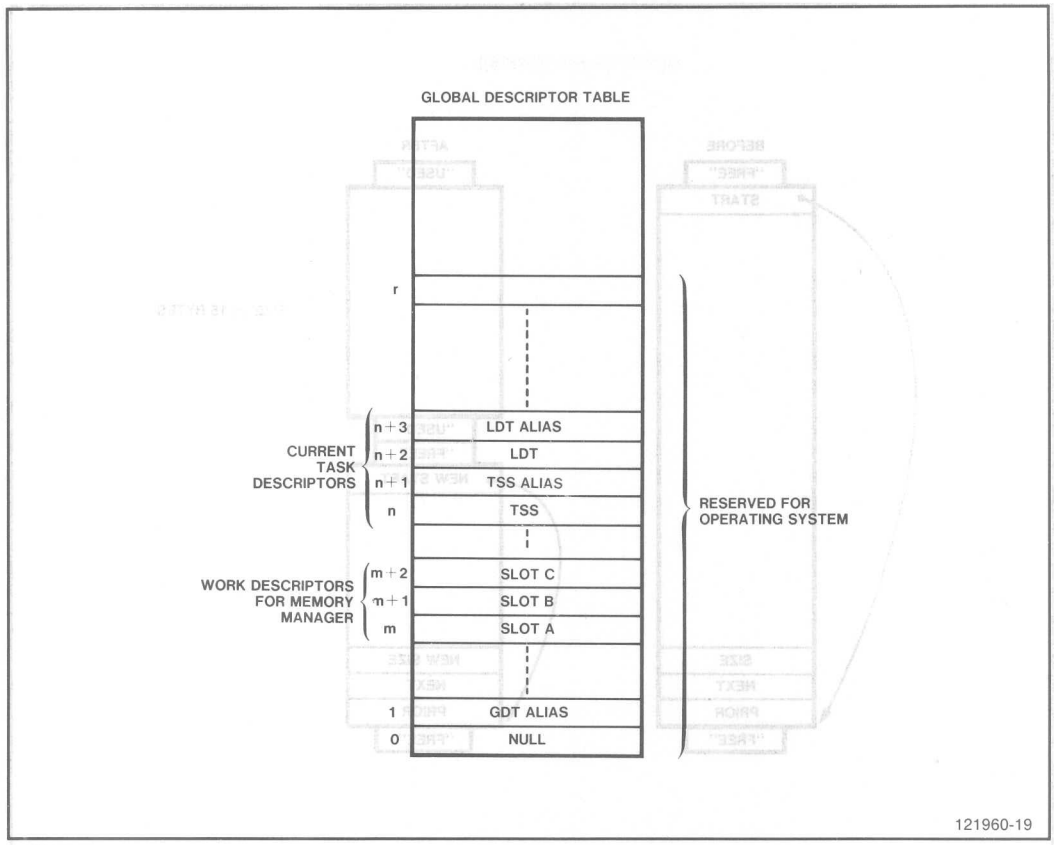


Figure 3-6. Example GDT Layout

When returning an unneeded segment to the available list, the procedure RETURN_SPACE checks the boundary tags of both the lower and higher adjacent segments to see whether there is another free segment with which to combine. Four cases are possible, as illustrated in figure 3-8 at the end of this chapter. Table 3-1 summarizes the actions taken in each of the four cases.

See figure 3-9 at the end of this chapter for the PL/M-286 code that implements this example of a memory manager.

Protection Structure

Where in the two-dimensional grid of protection offered by the iAPX 286 should the memory-management module lie? There are two approaches that offer different advantages:

1. The module can be structured as privileged procedures that execute as part of every task that calls them.
2. The module can execute as a separate task.

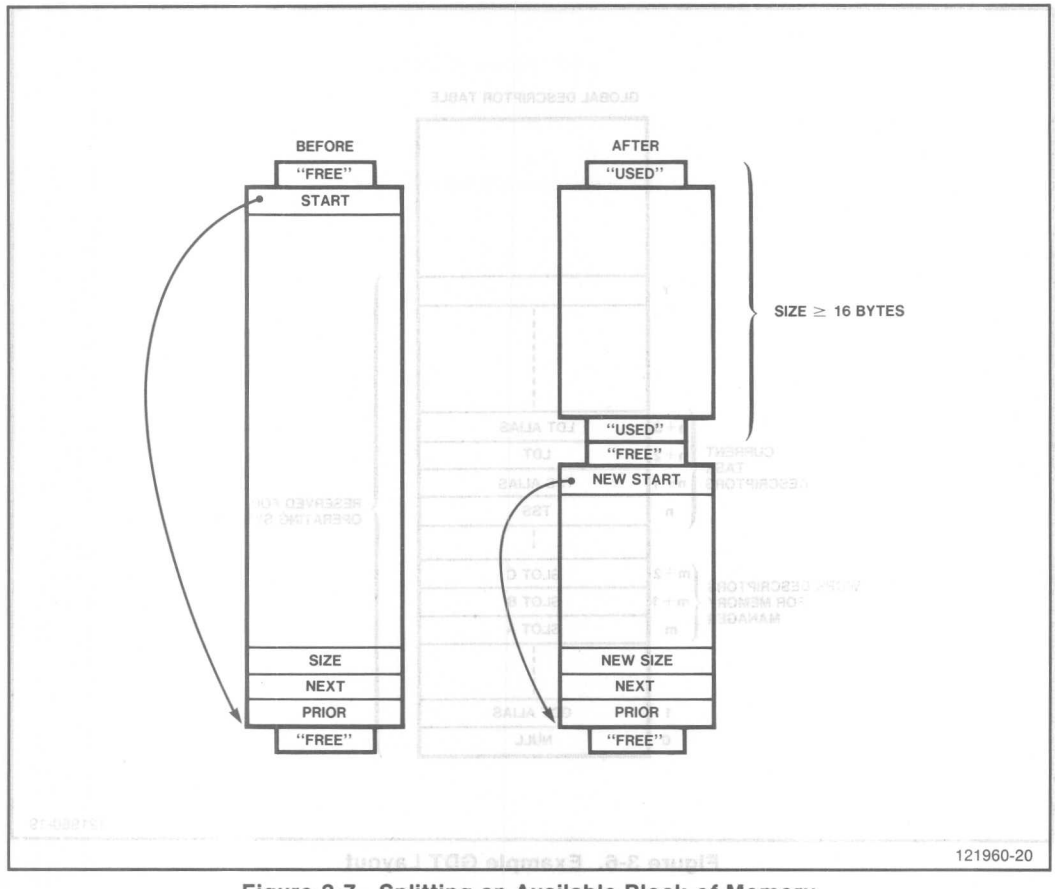


Figure 3-7. Splitting an Available Block of Memory

GLOBAL PROCEDURES

Placing the module's segment descriptors in the GDT allows all tasks in the system to share the module yet requires only one copy of the module's segments to be present in memory. This approach allows for fastest communication between the application and the memory manager.

The procedures of the memory manager synchronize with the calling procedure (that is to say, the calling procedure waits until the memory-management procedure returns). However, more than one task can be executing the memory-management procedures at one time. This can be an advantage (as when there are multiple CPUs and the requests are for different regions of the memory space), but it requires synchronization of changes to space-management data structures (not shown in this example).

Segments containing procedures and data structures internal to this most critical operating-system module should have the greatest protection possible. Because none of these procedures and data structures are PUBLIC, no other modules can gain knowledge of the locations of data and procedures. This by itself, however, does not constitute positive protection from accidental or intentional snooping or destruction. The segments containing these procedures and data should have privilege level 0 (PL 0) so that the processor can prevent any access from less trusted procedures.

Table 3-1. Actions for Combining Segments

Action	Case 0		Case 1		Case 2		Case 3	
	Lo Free	Hi Free	Lo Free	Hi Used	Lo Used	Hi Free	Lo Used	Hi Used
CURR_AVLBL=	Lo START		Lo START		This Base		This Base	
CURR_LINK.PRIOR=	—		—		Hi LINK.PRIOR		Null	
CURR_LINK.NEXT=	—		—		Hi LINK.NEXT		FIRST_AVLBL	
DELINK HI	Yes		No		No		No	
FIRST_AVLBL= CURR_AVLBL?	No		No		No		Yes	
PRIOR FREE LINK.NEXT= CURR_AVLBL?	No		No		Yes		No	
NEXT FREE LINK.NEXT= CURR_AVLBL?	No		No		Yes		Yes	

The PUBLIC interface procedures ALLOCATE and FREE_SEG should execute at PL 0 to access the internal data structures and procedures, which are also at PL 0. ALLOCATE and FREE_SEG should have gates at PL 3, however, so that even least trusted application procedures can get the space they need for such purposes as dynamic data structures.

SEPARATE TASK

Another possible structure, not exemplified here, is to treat the memory-management module as a separate task. The iAPX 286 features for isolation of tasks provide the needed protection for the critical memory-management structures and procedures. Within the memory-management task, privilege levels can be used to isolate the various internal procedures and data structures from one another. The memory-manager task can use a message passing mechanism such as that shown in Chapter 5 to receive requests and to pass segments to the requesting task.

With the memory-manager executing as a separate task, serialization of requests for space is automatic, thereby eliminating the need for synchronization when modifying space-management structures. The separate-task approach is also advantageous when there is a need to have the requesting task wait until sufficient memory becomes available to fullfil a request. While one task is waiting, the memory manager can continue to service requests from other tasks.

ADVANCED MEMORY MANAGEMENT

In actual applications, the memory-management module may need to deal with such topics as

- Different kinds of memory. The ALLOCATE procedure needs a parameter to specify (for example) slow versus fast memory, and each type of memory needs its own free-space list.
- Multiprocessing. When multiple processors share some, but not all, of the available memory, the memory management module must know what memory addresses each processor can access. ALLOCATE must provide memory that is accessible to the processor that is running the calling task. You need to partition memory into areas so that all the addresses in a given area satisfy a common accessibility constraint. (The criteria for partitioning may also include memory types as mentioned previously.) Each area needs its own free-space list.
- Dynamic deletion of memory. When a memory parity error occurs, the need for continued system operation may require deleting the affected block of memory from the available-space lists, so that it cannot cause more trouble.
- Fixed-location segments. Often certain addresses of memory have specific uses, for example, video refresh buffers, and communication blocks for intelligent peripheral controllers. The memory manager must be aware of these areas and not use them for other purposes. Its interfaces must include the means for a task to request a specific special-purpose area.
- Compaction. It can happen that no single free memory space is large enough to satisfy an ALLOCATE request, even though there is enough total free memory. Some memory-management subsystems call on a *compaction* algorithm in such cases. Whether implementation of a compaction algorithm is worthwhile depends primarily on the pattern of memory usage in a given application. In many applications, this situation arises only when memory is nearly full anyway; compaction in this case merely delays the inevitable by an insignificant time. If you do implement compaction, you may choose to associate with each allocated segment a pointer that helps the compaction algorithm find the descriptors for that segment. With a memory-management scheme such as that exemplified here, the boundary tags are the most convenient container for descriptor pointers. With descriptor pointers in place, the compaction algorithm need only follow the available-space lists to discover all the opportunities for compaction.

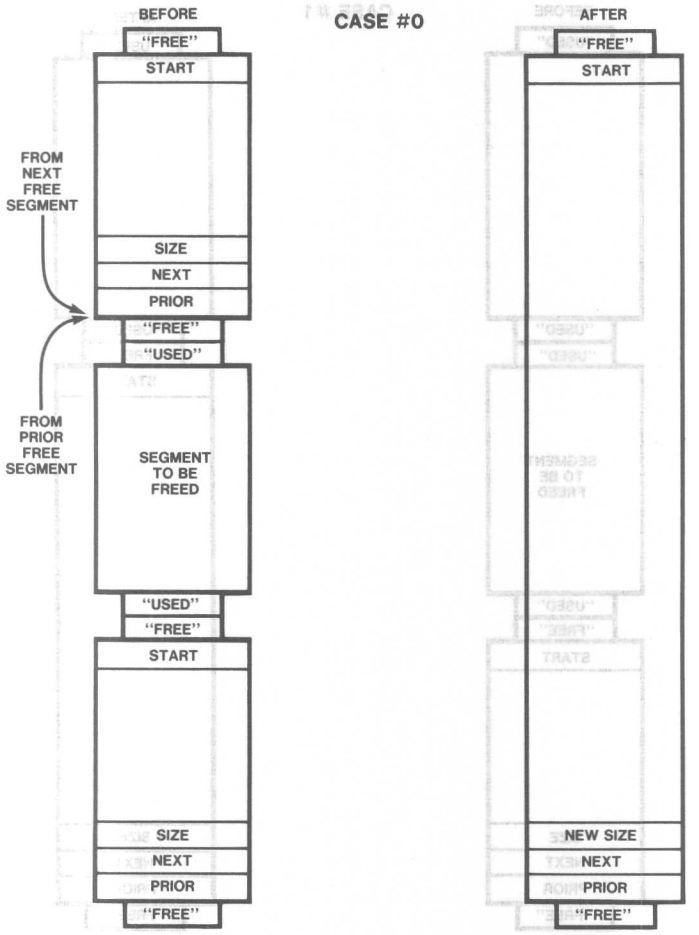
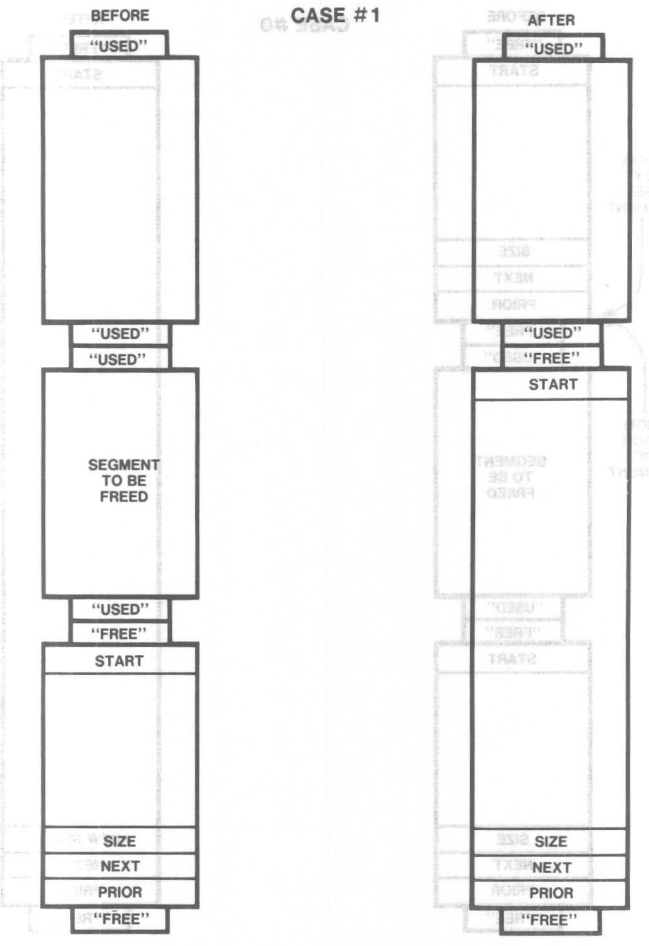


Figure 3-8. Possibilities for Combining Segments



121960-21

Figure 3-8. Possibilities for Combining Segments (Cont'd.)

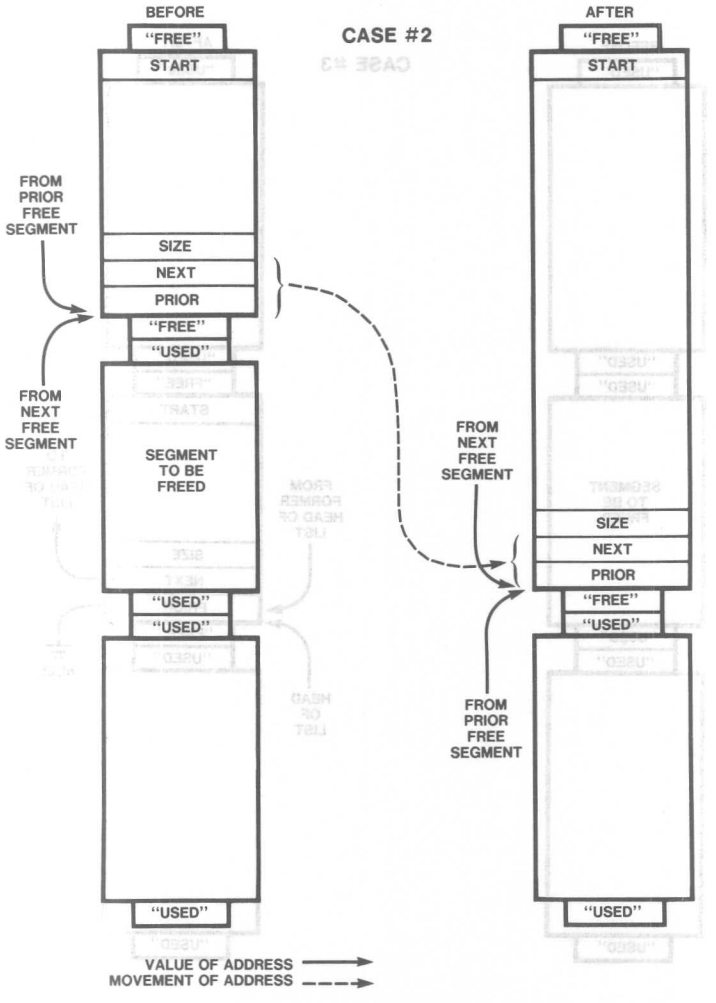
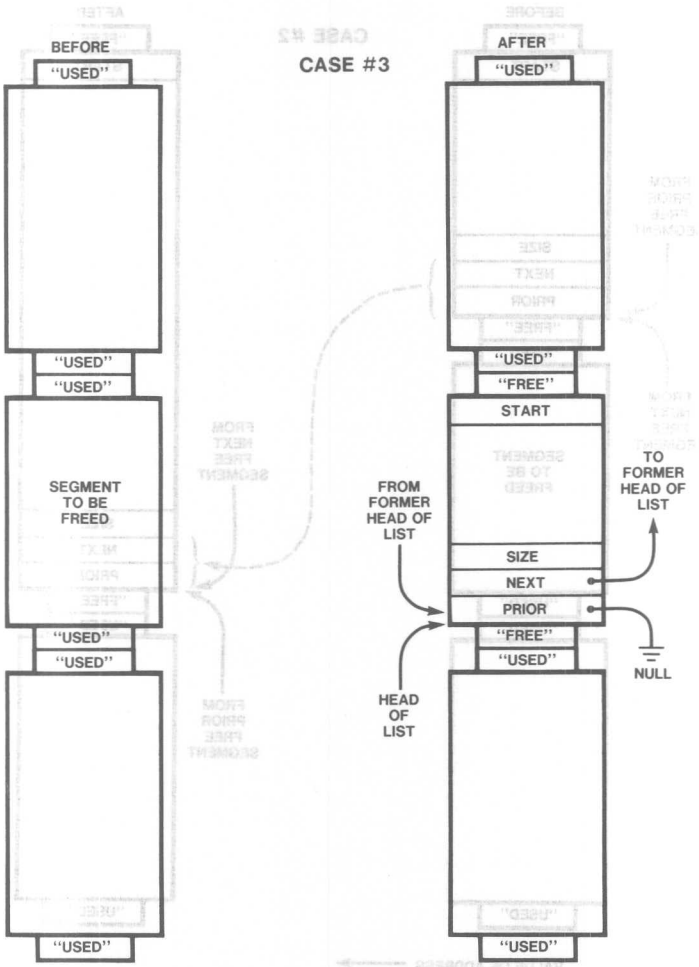


Figure 3-8. Possibilities for Combining Segments (Cont'd.)



121960-21

Figure 3-8. Possibilities for Combining Segments (Cont'd.)

```
PL/M-286 COMPILER     960-501                    date          PAGE  1


system-ID  PL/M-286 DEBUG Vx.y COMPILATION OF MODULE MEMORY
OBJECT MODULE PLACED IN :F1:MEMORY.OBJ
COMPILER INVOKED BY:  :F3:PLM286.86 :F1:MEMORY.PLM CODE DEBUG


            $ PAGEWIDTH(71) TITLE('960-501') INCLUDE (:F1:NUCSUB.PLM)
          = $ NOLIST

   1        MEMORY:  DO;

            /*****************************************************/
            /*                    Externals.                     */
   2   1    POINT_AT: PROCEDURE (SLOT, RIGHTS, PHYS_ADDR_PTR, LIMIT)
                                                  EXTERNAL;
   3   2     DECLARE SLOT SELECTOR, RIGHTS BYTE,
                     PHYS_ADDR_PTR POINTER, LIMIT WORD;
   4   2     END POINT_AT;
   5   1    NULLIFY: PROCEDURE (SLOT) EXTERNAL;
   6   2     DECLARE SLOT SELECTOR;
   7   2     END NULLIFY;
   8   1    GETSEGMENTBASE: PROCEDURE (SEL, BASE_ADDR_PTR) EXTERNAL;
   9   2     DECLARE SEL SELECTOR, BASE_ADDR_PTR POINTER;
  10   2     END GETSEGMENTBASE;

            /*****************************************************/
            /*          Space-management definitions.            */
  11   1    DECLARE PARAGRAPH LITERALLY '16';
             /* To run under SIM286, all segments must have a base
                address equal to zero mod PARAGRAPH.  This is not
                required when running on iAPX 286 hardware.      */

  12   1    DECLARE MEM_LINK LITERALLY
             'PADDING (8) BYTE,
              START   DWORD,
              HI_TAG  WORD,
              LO_TAG  WORD,
              PRIOR   DWORD,
              NEXT    DWORD,
              SIZE    DWORD';
             /* Base address of link descriptor is always PARAGRAPH
                less than address of PRIOR field.  Address of PRIOR
                field is always 0 mod PARAGRAPH.  PRIOR, NEXT, and
                SIZE fields always point to a PRIOR - PARAGRAPH
                address. */

  13   1    DECLARE TAGS_SIZE  LITERALLY '4';
             /* The space used by both tag words */

  14   1    DECLARE LINK_LIMIT LITERALLY '27';
             /* Limit used to construct descriptors for MEM_LINK */

  15   1    DECLARE BLOCK_MODULUS LITERALLY '32';
             /* All memory blocks are an integral multiple of
                BLOCK_MODULUS in length.                         */
```

Figure 3-9. Code for Memory-Management Example

```
PL/M-286 COMPILER     960-501                 date          PAGE   2


  16   1       DECLARE NULL_PHYS_ADDR LITERALLY '0';

  17   1       DECLARE USED        LITERALLY '1',
                       FREE        LITERALLY '0';
                /* Values of boundary tags */
  18   1       DECLARE OK          LITERALLY '0',
                       FAILED      LITERALLY '8000H';
                /* Values of exception codes */

  19   1       DECLARE DWRIGHTS    LITERALLY '92H' /* For manipulating
                        space-management data structures, this module
                        needs these rights parameters: Present, DPL=0,
                        data segment, grow up, writable.    */ ;

                /*******************************************************/
                /*          Space-management data structures.          */

  20   1       DECLARE (FIRST_AVLBL,LAST_AVLBL,CURR_AVLBL)DWORD PUBLIC;
                /* Physical-address pointers to chain of available
                   space.  These always point to PRIOR - PARAGRAPH to
                   avoid calculating base addresses for MEM_LINKs.   */

  21   1       DECLARE SLOT_A  SELECTOR PUBLIC,
                       WSLOT_A WORD AT (@SLOT_A) INITIAL (38H), /* 7 */
                       SLOT_B  SELECTOR PUBLIC,
                       WSLOT_B WORD AT (@SLOT_B) INITIAL (40H), /* 8 */
                       SLOT_C  SELECTOR PUBLIC,
                       WSLOT_C WORD AT (@SLOT_C) INITIAL (48H); /* 9 */
                /* "Scratch" slots for addressing MEM_LINKS.          */
                /* Be sure to reserve these slots with the Builder    */

                /*******************************************************/
                /*   Round a size parameter upwards to next greater    */
                /* or equal (N * BLOCK_MODULUS) - TAGS_SIZE for some N */

  22   1       ROUND_SIZE: PROCEDURE (A_PTR) PUBLIC REENTRANT;

  23   2        DECLARE A_PTR POINTER,
                       ADDR BASED A_PTR DWORD;

  24   2        ADDR = BLOCK_MODULUS
                  * (((ADDR + TAGS_SIZE - 1)  / BLOCK_MODULUS) + 1)
                  - TAGS_SIZE;
  25   2        END ROUND_SIZE;

                /*******************************************************/
                /*          Delink from available space list.          */

  26   1       DELINK:  PROCEDURE (THIS_SEL) REENTRANT;
  27   2        DECLARE THIS_SEL   SELECTOR,
                       THIS_LINK  BASED THIS_SEL STRUCTURE (MEM_LINK);

  28   2        DECLARE PRIOR_LINK BASED SLOT_C  STRUCTURE (MEM_LINK),
```

Figure 3-9. Code for Memory-Management Example (Cont'd.)

```
PL/M-286 COMPILER     960-501                 date            PAGE   3


                    NEXT_LINK  BASED SLOT_C  STRUCTURE (MEM_LINK);

  29   2        IF THIS_LINK.PRIOR = NULL_PHYS_ADDR
                 /* This is the beginning of the list. */
  30   2         THEN FIRST_AVLBL = THIS_LINK.NEXT ;
  31   2         ELSE DO; /* Update link from prior segment. */
  32   3          CALL POINT_AT(SLOT_C, DWRIGHTS,
                                  @THIS_LINK.PRIOR, LINK_LIMIT);
  33   3          PRIOR_LINK.NEXT = THIS_LINK.NEXT;
  34   3         END;
  35   2        IF THIS_LINK.NEXT = NULL_PHYS_ADDR
                 /* This is the end of the list. */
  36   2         THEN  LAST_AVLBL = THIS_LINK.PRIOR;
  37   2         ELSE DO; /* Update link from next segment. */
  38   3          CALL POINT_AT(SLOT_C, DWRIGHTS,
                                  @THIS_LINK.NEXT, LINK_LIMIT);
  39   3          NEXT_LINK.PRIOR = THIS_LINK.PRIOR;
  40   3         END;

  41   2       END DELINK;

                /*****************************************************/
  42   1       FIND_FIRST_FIT: PROCEDURE (SIZE,BASE_ADDR_PTR)
                                          WORD REENTRANT;

  43   2        DECLARE SIZE          WORD,
                        BASE_ADDR_PTR POINTER,
                        BASE_ADDR BASED BASE_ADDR_PTR DWORD;

  44   2        DECLARE CURR_LINK  BASED SLOT_A STRUCTURE (MEM_LINK),
                        PHYS_SIZE  DWORD,
                        SIZE_DIFF  DWORD,
                         /* Boundary tag items */
                        BOUND_ADDR DWORD,
                        BOUND_MID  BASED SLOT_B STRUCTURE (MEM_LINK),
                        BOUND_HI   BASED SLOT_B STRUCTURE (MEM_LINK);

  45   2        DECLARE TOP_LOOP LABEL;

  46   2        PHYS_SIZE = SIZE;
  47   2        CALL ROUND_SIZE(@PHYS_SIZE);
  48   2        CALL POINT_AT(SLOT_A, DWRIGHTS,
                                  @CURR_AVLBL, LINK_LIMIT);

  49   2       TOP_LOOP:
                IF SLOT_A = SELECTOR$OF(NIL) /* Check for end of list */
  50   2         THEN DO;
  51   3          IF FIRST_AVLBL = NULL_PHYS_ADDR
                   /* The list is empty. */
  52   3           THEN RETURN FAILED;
  53   3         END;
  54   2         ELSE CALL POINT_AT(SLOT_A, DWRIGHTS,
                                      @FIRST_AVLBL, LINK_LIMIT);
                  /* Continue from beginning of list. */

  55   2        IF CURR_LINK.SIZE < PHYS_SIZE
                 THEN /* This segment is too small, so... */
```

Figure 3-9. Code for Memory-Management Example (Cont'd.)

```
PL/M-286 COMPILER    960-501                 date                PAGE   4


  56   2          DO; /* Look at next free segment in the list. */
  57   3          CALL POINT_AT(SLOT_A, DWRIGHTS,
                                  @CURR_LINK.NEXT, LINK_LIMIT);
  58   3          IF CURR_AVLBL = CURR_LINK.NEXT
                   /* Have searched entire list without a hit. */
  59   3           THEN DO;
  60   4            CALL NULLIFY(SLOT_A);
  61   4            RETURN FAILED;
  62   4           END;
  63   3          GOTO TOP_LOOP;
  64   3          END; /* Segment too small. */

  65   2        SIZE_DIFF = CURR_LINK.SIZE - PHYS_SIZE;
                 /* Always a multiple of BLOCK_MODULUS */

  66   2        IF SIZE_DIFF = 0
  67   2         THEN DO; /* This segment is a close fit. */
  68   3          CURR_AVLBL = CURR_LINK.NEXT;
  69   3          CALL DELINK (SLOT_A);

                  /* Set lower boundary tag */
  70   3          CURR_LINK.LO_TAG = USED;
                  /* Set upper boundary tag */
  71   3          CALL GET$SEGMENT$BASE (SLOT_A, @BOUND_ADDR);
  72   3          BASE_ADDR = BOUND_ADDR;
  73   3          BOUND_ADDR = BOUND_ADDR + CURR_LINK.SIZE + TAGS_SIZE;
  74   3          CALL POINT_AT(SLOT_B,DWRIGHTS,@BOUND_ADDR,LINK_LIMIT);
  75   3          BOUND_HI.HI_TAG =USED;

  76   3          CALL NULLIFY(SLOT_A); CALL NULLIFY(SLOT_B);
  78   3          BASE_ADDR = BASE_ADDR + PARAGRAPH;
  79   3          RETURN OK;
  80   3         END; /* Close fit. */

  81   2         ELSE DO; /* It will fit here with room to spare. */
  82   3          CALL GET$SEGMENT$BASE (SLOT_A, @CURR_AVLBL);
                   /* Set boundary tag at end of new segment. */
  83   3          BOUND_ADDR = CURR_AVLBL + CURR_LINK.SIZE + TAGS_SIZE;
  84   3          CALL POINT_AT(SLOT_B,DWRIGHTS,@BOUND_ADDR,LINK_LIMIT);
  85   3          BOUND_HI.HI_TAG = USED;
                   /* Calculate starting address of the new segment. */
  86   3          BOUND_ADDR = CURR_AVLBL + SIZE_DIFF;
  87   3          BASE_ADDR  = BOUND_ADDR + PARAGRAPH;
                   /* Set the boundary fields between the 2 segments. */
  88   3          CALL POINT_AT(SLOT_B,DWRIGHTS,@BOUND_ADDR,LINK_LIMIT);
  89   3          BOUND_MID.START = CURR_AVLBL;
  90   3          BOUND_MID.HI_TAG = FREE;
  91   3          BOUND_MID.LO_TAG = USED;
                   /* Change size of available segment,
                      considering the 2 boundary tag words. */
  92   3          CURR_LINK.SIZE = SIZE_DIFF - TAGS_SIZE;
  93   3          CALL NULLIFY(SLOT_A); CALL NULLIFY(SLOT_B);
  95   3          RETURN OK;
  96   3         END; /* Room to spare. */

  97   2       END FIND_FIRST_FIT;
               $EJECT
```

Figure 3-9. Code for Memory-Management Example (Cont'd.)

```
PL/M-286 COMPILER     960-501                 date          PAGE    5


                /*******************************************************/
                /*   Place a segment into the available space list.    */

  98    1       RETURN_SPACE: PROCEDURE(FREE_SEG) REENTRANT;

  99    2        DECLARE FREE_SEG                    SELECTOR,
                        FREE_LINK  BASED FREE_SEG  STRUCTURE (MEM_LINK);

 100    2        DECLARE CURR_LINK  BASED SLOT_A STRUCTURE (MEM_LINK),
                        NEIBR_ADDR          DWORD,
                        NEIBR_LINK BASED SLOT_B STRUCTURE (MEM_LINK),
                        FREE_BASE           DWORD,
                        FREE_SIZE           DWORD,
                        HI_SIZE             DWORD,
                        NCASE               BYTE,
                         HI_USED LITERALLY '01H',
                         LO_USED LITERALLY '02H';

 101    2       CALL GET$SEGMENT$BASE (FREE_SEG, @FREE_BASE);
 102    2       FREE_BASE = FREE_BASE - PARAGRAPH; /* point to tags */
 103    2       FREE_SIZE = GET$SEGMENT$LIMIT (FREE_SEG);
 104    2       FREE_SIZE = FREE_SIZE + 1;
 105    2       CALL ROUND_SIZE (@FREE_SIZE);
                /* Determine which case. */
 106    2       NCASE = 0;
                /* Check higher neighbor. */
 107    2       NEIBR_ADDR = FREE_BASE + FREE_SIZE + TAGS_SIZE;
 108    2       CALL POINT_AT (SLOT_B,DWRIGHTS,@NEIBR_ADDR,LINK_LIMIT);
 109    2       IF NEIBR_LINK.LO_TAG = USED THEN NCASE=NCASE OR HI_USED;
 111    2        ELSE HI_SIZE = NEIBR_LINK.SIZE; /* Save */
                /* Check lower neighbor. */
 112    2       NEIBR_ADDR = FREE_BASE;
 113    2       CALL POINT_AT(SLOT_B,DWRIGHTS,@NEIBR_ADDR,LINK_LIMIT);
 114    2       IF NEIBR_LINK.HI_TAG=USED THEN NCASE=NCASE OR LO_USED;

                /* Which segment should become the new current one? */
 116    2       IF (NCASE AND LO_USED)<>0
 117    2        THEN CURR_AVLBL = FREE_BASE; /* low neighbor used */
 118    2        ELSE CURR_AVLBL = NEIBR_LINK.START; /* low free */
 119    2       CALL POINT_AT(SLOT_A,DWRIGHTS,@CURR_AVLBL,LINK_LIMIT);

                /* Calculate size of new segment. */
 120    2       IF (NCASE AND LO_USED) = LO_USED /* if low neibr used */
 121    2        THEN CURR_LINK.SIZE = 0;
 122    2        ELSE /* already contains size of low neighbor */
                  CURR_LINK.SIZE = CURR_LINK.SIZE + TAGS_SIZE;
 123    2       IF (NCASE AND HI_USED) <> HI_USED
                  /* if high neighbor free */
 124    2        THEN CURR_LINK.SIZE = CURR_LINK.SIZE+HI_SIZE+TAGS_SIZE;
 125    2       CURR_LINK.SIZE = CURR_LINK.SIZE + FREE_SIZE;

                /* Set next and prior links in new segment. */
 126    2       IF NCASE = 3 /* neither neighbor free */
 127    2        THEN DO; /* insert at head of available list */
 128    3         CURR_LINK.PRIOR = NULL_PHYS_ADDR;
 129    3         CURR_LINK.NEXT = FIRST_AVLBL;
 130    3        END;
```

Figure 3-9. Code for Memory-Management Example (Cont'd.)

```
PL/M-286 COMPILER     960-501                 date            PAGE   6


 131   2        ELSE IF (NCASE AND HI_USED) <> HI_USED
                   /* if high neighbor free */
 132   2         THEN DO; /* Dispose with high neighbor's links. */
                   /* Make selector for high neighbor. */
 133   3          NEIBR_ADDR = FREE_BASE + FREE_SIZE + TAGS_SIZE;
 134   3          CALL POINT_AT(SLOT_B,DWRIGHTS,@NEIBR_ADDR,LINK_LIMIT);
 135   3          IF NCASE = 0 /* both neighbors free */
                     THEN /* remove one from available list. */
 136   3            CALL DELINK (SLOT_B);
 137   3            ELSE /* Must be case 2.*/
                     DO /* Transfer links to new current. */;
 138   4              CURR_LINK.PRIOR = NEIBR_LINK.PRIOR;
 139   4              CURR_LINK.NEXT  = NEIBR_LINK.NEXT;
 140   4             END;
 141   3         END; /* disposing with high neighbor's links. */

 142   2        IF (NCASE AND LO_USED) = LO_USED
                   /* if low neighbor used. */
 143   2         THEN DO; /* Fix up links in prior and next segments. */
 144   3          IF CURR_LINK.PRIOR = NULL_PHYS_ADDR
                    THEN /* there is no prior */
 145   3            FIRST_AVLBL = CURR_AVLBL;
 146   3           ELSE DO; /* fix up prior */
 147   4            NEIBR_ADDR = CURR_LINK.PRIOR;
 148   4            CALL POINT_AT (SLOT_B, DWRIGHTS, @NEIBR_ADDR,
                                   LINK_LIMIT);
 149   4            NEIBR_LINK.NEXT = CURR_AVLBL;
 150   4           END;
 151   3          IF CURR_LINK.NEXT = NULL_PHYS_ADDR
                    THEN /* there is no next */
 152   3            LAST_AVLBL = CURR_AVLBL;
 153   3           ELSE DO; /* fix up next */
 154   4            NEIBR_ADDR = CURR_LINK.NEXT;
 155   4            CALL POINT_AT (SLOT_B, DWRIGHTS, @NEIBR_ADDR,
                                   LINK_LIMIT);
 156   4            NEIBR_LINK.PRIOR = CURR_AVLBL;
 157   4           END;
 158   3         END; /* Fixing up links. */

                   /* Set tag words */
 159   2         CURR_LINK.LO_TAG = FREE;
                   /* Set START field in new current segment. */
 160   2         NEIBR_ADDR = CURR_AVLBL + CURR_LINK.SIZE + TAGS_SIZE;
 161   2         CALL POINT_AT(SLOT_B,DWRIGHTS,@NEIBR_ADDR,LINK_LIMIT);
 162   2         NEIBR_LINK.START = CURR_AVLBL;
 163   2         NEIBR_LINK.HI_TAG = FREE;
 164   2         CALL NULLIFY (SLOT_A); CALL NULLIFY (SLOT_B);

 166   2       END RETURN_SPACE;
              $EJECT
```

Figure 3-9. Code for Memory-Management Example (Cont'd.)

```
PL/M-286 COMPILER    960-501                 date            PAGE   7


                /*******************************************************/

 167   1        ALLOCATE: PROCEDURE (SLOT, RIGHTS, SIZE, EXCEP_PTR)
                                                 PUBLIC REENTRANT;

 168   2         DECLARE SLOT       SELECTOR,
                       RIGHTS     BYTE,
                       SIZE       WORD,
                       EXCEP_PTR  POINTER,
                       EXCEP BASED EXCEP_PTR WORD;

 169   2         DECLARE BASE_ADDR  DWORD;

 170   2         IF OK <> FIND_FIRST_FIT (SIZE, @BASE_ADDR)
 171   2          THEN DO;
 172   3            EXCEP = FAILED;
 173   3            RETURN;
 174   3           END;
 175   2         CALL POINT_AT (SLOT, RIGHTS, @BASE_ADDR, SIZE-1);
 176   2         EXCEP = OK;

 177   2        END ALLOCATE;

                /*******************************************************/

 178   1        FREE_SEG: PROCEDURE (SLOT, EXCEP_PTR) PUBLIC REENTRANT;

 179   2         DECLARE SLOT       SELECTOR,
                       EXCEP_PTR  POINTER,
                       EXCEP BASED EXCEP_PTR WORD;

 180   2         CALL RETURN_SPACE (SLOT);
 181   2         CALL NULLIFY (SLOT);
 182   2         EXCEP = OK;

 183   2        END FREE_SEG;

                /*******************************************************/

 184   1        END MEMORY;
```

Figure 3-9. Code for Memory-Management Example (Cont'd.)

```
END MEMORY;

\*****************************************************************\

END FREE_SEG;

 EXCEP = OK;
 CALL UNFLIP (SEG#);
 CALL RETURN_SPACE (SEG#);

          EXCEP_BASED EXCEP_PTR MODE
          EXCEP_PTR   POINTER
 DECLARE SEG#         SELECTOR

FREE_SEG: PROCEDURE (SEG#, EXCEP_PTR) PUBLIC;

\*****************************************************************\

END ALLOCATE;

 EXCEP = OK;
 CALL POINT_AT (SEG#, RIGHTS, BASE_ADDR, S
  END;
    RETURN;
    EXCEP = FAILED;
  THEN DO;
 IF OK <> FIND_FIRST_FIT (SIZE, BASE_ADDR)

 DECLARE BASE_ADDR  DWORD;

          EXCEP_BASED EXCEP_PTR MODE
          EXCEP_PTR   POINTER
          SIZE        WORD
          RIGHTS      BYTE
 DECLARE SEG#         SELECTOR

                                        PUBLIC REENT;
ALLOCATE: PROCEDURE (SEG#, RIGHTS, SIZE, EX

\*****************************************************************\
```

4 Task Management

CHAPTER 4
TASK MANAGEMENT

The primary responsibility of an operating system is to allocate the processor to the executing tasks so that each task makes progress consistent with its role in the application. This chapter examines how the task-oriented features of the iAPX 286 hardware apply to conventional task-management concepts.

HARDWARE TASK-MANAGEMENT FEATURES

The operating system's responsibility for managing a multitasking system is reduced by iAPX 286 features for saving and restoring task state and switching between tasks.

Storing Task State

The state of a task (from the processor's point of view) is the contents of the registers used by that task. The architecture of the iAPX 286 defines a special type of segment, the task state segment (TSS), for storing the 80286-related state of a task. Multitasking operating systems on any processor need to store similar information. The iAPX 286 requires merely that a specific format be used so that the CPU can store and restore task state automatically. Figure 4-1 illustrates a TSS and related hardware structures.

The processor keeps the location of the TSS of the currently executing task in the task register. The task register has two parts:

- The "visible" portion, which a task can access. This part contains a selector to the descriptor (in the GDT) for the current TSS.
- The "invisible" portion, which tasks do not control. When the contents of the visible portion are changed, the processor loads the invisible portion with the base and limit values from the TSS descriptor indexed by the selector in the visible portion.

There are two ways to change the task register:

- By one of the task switching operations described later in this section.
- By the LTR instruction. LTR is used to give the task register its initial value during system initialization. Only privilege-level 0 (PL-0) procedures can execute LTR.

Because TSSs correspond one-to-one with tasks, the selector of a TSS uniquely identifies a task. The STR instruction reads the task register into a selector. Operating-system procedures can use STR to identify the calling task. The built-in variable TASK$REGISTER gives PL/M-286 programs access to the task register.

The items in the TSS fall into four classes:

- Back link. Contains a selector to the TSS of the calling (or interrupted) task (if any).
- LDT selector. Contains a selector to this task's LDT.

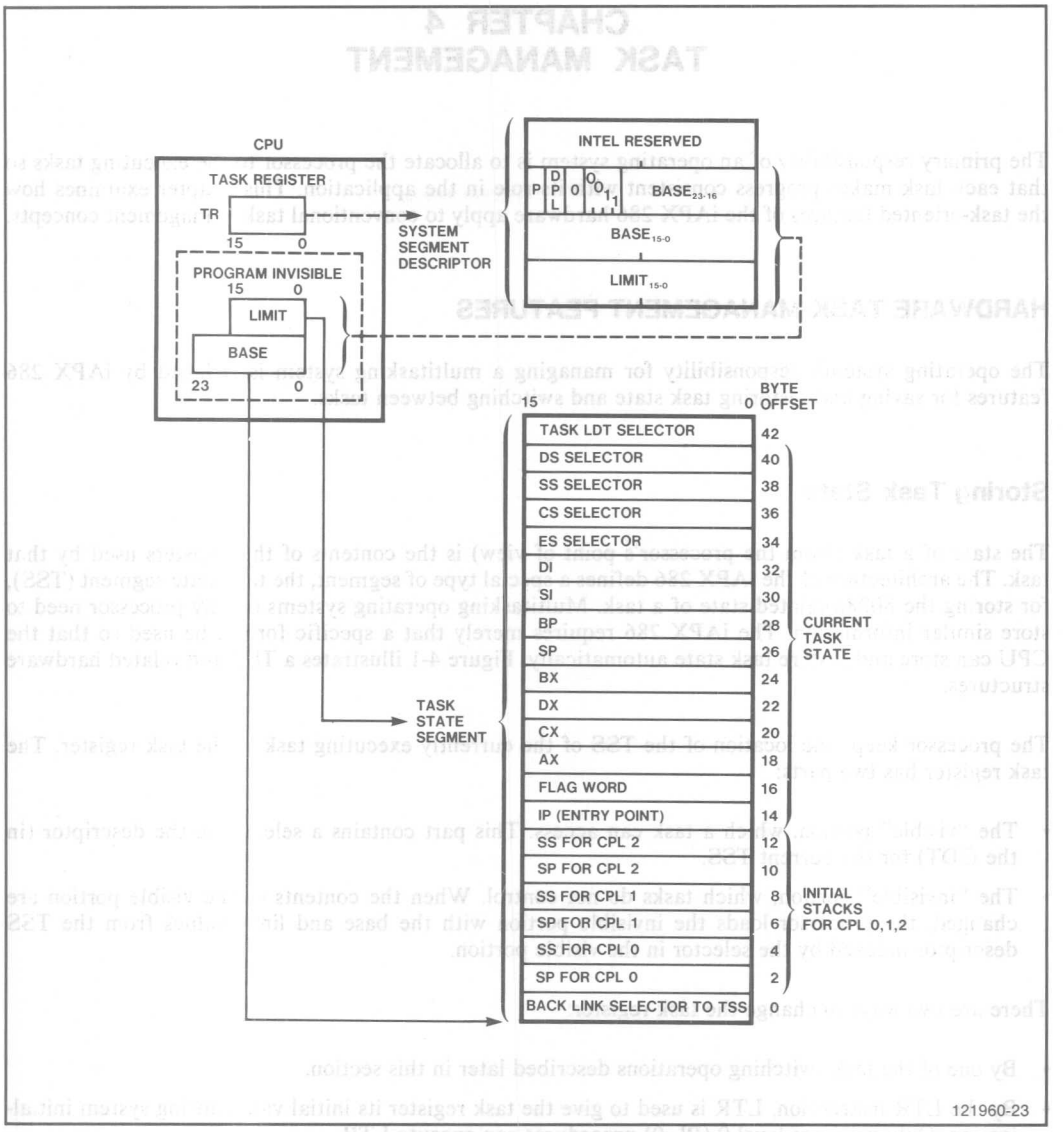


Figure 4-1. Task State Segment and Register

- Processor registers and flags. Used to store the processor state of the task. Note in particular the NT flag, which indicates when the BACK LINK contains a valid selector.

- Initial stacks. Contain the initial SS and SP values to be used when a CALL transfers control to any of the higher PLs (0, 1, or 2). No stack pointer is needed in the TSS for the PL-3 stack because that stack is either the current stack (pointed to by the SS:SP fields) or is locatable via the chain of stack pointers in higher-level stacks.

The processor updates only the back link, the registers, and the flags as part of a task switching operation. The processor merely reads the initial stack fields (during interlevel CALLs) and the LDT selector (during a task switch). The operating system is responsible for initializing the stack and LDT fields. It must also update the LDT selector before changing the LDT register.

Switching Tasks

Since a multitasking system typically has more tasks than it has processors to execute those tasks, there must be some provision for causing a processor to cease executing one task and begin executing another. The 80286 has several such mechanisms, each appropriate to different situations. All use ordinary JMP, CALL, INT, or IRET instructions. The destination operand determines whether a task switch occurs. Table 4-1 summarizes the operations and operands that cause switching of tasks.

In PL/M-286, an indirect CALL statement to the selector of a TSS descriptor or task gate causes generation of an intertask CALL instruction. The WAITFORINTERRUPT built-in procedure generates an IRET instruction.

The operand of a CALL or JMP instruction is a pointer containing both selector and offset parts; in this case, the offset is not used, however. The selector portion may refer either to the descriptor of a TSS or to a task gate for a TSS. The result is the same in either case. The difference lies in the protection of access to the TSS. TSS descriptors, which may reside only in the GDT, normally have DPL set to zero to prevent unauthorized task switching by procedures outside the operating system. Task gates may reside in any descriptor table, giving task switching ability to procedures that have access to the gate. A task gate in an LDT gives task switching power only to that LDT's task. A task gate in the IDT gives task switching power to interrupts. This reduces operating-system involvement in interrupt handling, and thereby reduces the time needed to respond to interrupts.

The operating system normally uses a JMP instruction to a TSS descriptor to cause a task switch. A CALL instruction to a task gate in an LDT is useful for implementing coroutines.

The IRET instruction exits from a nested task that is invoked by a CALL as well as from one invoked by an interrupt. (You cannot use the RET instruction to exit from a CALLed task; RET does not perform a task switch.) The NT (nested task) flag controls the function of the IRET instruction. When a CALL (or interrupt) causes a task switch, the processor sets the NT flag as the new task begins and fills the back-link of the new TSS with the selector of the calling task's TSS. When the NT flag is set, the IRET instruction causes a task switch to the task whose TSS selector is stored in the back-link. The new task must be marked as busy (type code = 3); otherwise an exception occurs.

Table 4-1. Task Switching Operations

Operation	Descriptor Referenced	Descriptor Table
CALL, JMP, IRET	TSS	GDT
CALL, JMP	task gate	GDT, LDT
INT instruction, external interrupt, exception	task gate	IDT

If the processer, while executing an IRET instruction, finds the NT flag not set, this indicates a return to an interrupted procedure in the *same* task. Refer to Chapter 6 for details on interrupt procedures.

For CALL and JMP instructions, success of a task switch operation depends on the type field of the TSS descriptor. If the type code is 1, denoting an available task, then the task switch proceeds. If the type code is 3, denoting a busy task, then an attempt to switch to the task causes an exception. A task is busy under either of these conditions:

- The task is the currently executing task.
- The task is on the chain of TSS back-links from the currently executing task. This prevents recursion of task invocations. A task should be restarted only by the the task that interrupts it or by the operating system after the task has been removed from the back-link chain.

If the target task is not busy, the processor takes these steps in executing a task switch:

- Saves all registers in current TSS
- Loads TR with new TSS descriptor
- Loads all registers and flags from new TSS (including LDT register)
- If switch is due to CALL (or interrupt), sets NT flag and sets back-link in new TSS to point to previous TSS
- If switch is due to JMP or IRET, changes the old task's descriptor type code to one, indicating that the task is no longer busy
- Resumes new task where it left off (i.e., CS:IP from new TSS)

Note that you cannot pass parameters by an intertask CALL. It is possible to share data between tasks, however. Chapter 5 takes up this subject.

ROLE OF OPERATING SYSTEM IN TASK MANAGEMENT

Task switching without operating-system involvement is possible (though not necessarily advisable) in static systems. Consider the following two application-driven scheduling strategies for static systems:

1. A fixed sequence of tasks is defined, and each task, when ready to relinquish the processor, voluntarily calls or jumps to the next task in sequence. Barring any errors, each task gets a share of processor time.
2. All tasks in the system service external events. The interrupt mechanism of the iAPX 286, by means of interrupt tasks, causes task initiation in real-time response to those events.

Strategy 1 is not viable in a highly protected system. Errors do happen. An erroneous program might easily skip a task entirely. A programming error that causes a tight loop in one task would prevent all other tasks from being serviced.

Strategy 2 can be adequate by itself for certain real-time systems with a static mix of tasks. Task switching by interrupt is usable in dynamic systems, too, but rarely do all tasks in a dynamic system deal exclusively with interrupts. Therefore, in dynamic systems, in highly protected systems, and in systems with tasks that do not provide real-time processing, the operating system may need to assist the processor with task switching.

State Model of Task Scheduling

The role that the operating system must play in using the iAPX 286 task features is most conveniently expressed in terms of a state-transition model. To distinguish from the processing state of a task (as stored in the TSS), the term *scheduling state* is used here. Figure 4-2 illustrates the scheduling states that a task may assume and the events that may cause a change of state.

A RUNNING task is the one that the processor is executing. A RUNNING task continues to execute until

- It voluntarily gives up control because it needs to wait for some event, such as completion of an I/O operation, data from another program, or the end of a specific period of time.
- It is preempted, i.e., forced to give up control. The interrupt mechanism may cause preemption in order to execute an interrupt task, or the operating system may preempt periodically (via timer interrupt) to give another task a chance to receive its share of the processor's attention.

A WAITING task may be waiting for any of several events:

- Completion of a request to the OS for I/O
- A signal from another task
- Data from another task
- The end of a time-delay period

The READY state is really a special case of the WAITING state. A READY task is waiting for one specific event: the availability of a processor to execute it. A task becomes READY when first created. A WAITING task becomes READY upon occurrence of the event (or events) for which it is waiting. A RUNNING task becomes READY when preempted.

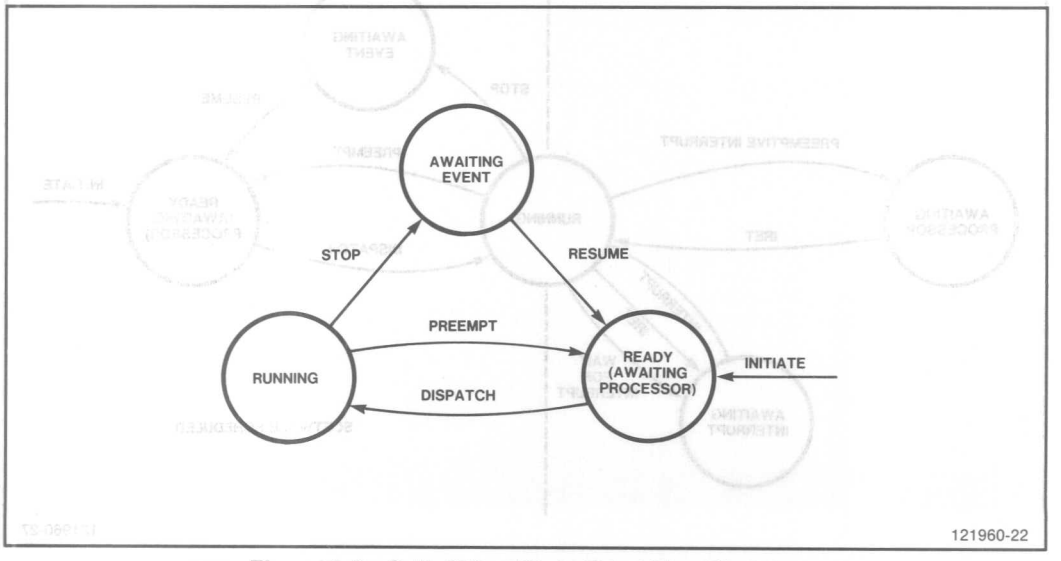


Figure 4-2. Scheduling State Transition Diagram

Usually, termination of a task is possible regardless of its scheduling state; therefore, this diagram does not illustrate the transition to "terminated" state.

Interfacing with the Hardware Scheduler

Many applications of the iAPX 286 need both software scheduling (by the operating system) and hardware scheduling (by the interrupt mechanism), but when two schedulers work with the same set of tasks, you must ensure that they work together harmoniously. Figure 4-3 illustrates the additional complexity of dual schedulers.

Note that scheduling state under hardware scheduling is nearly analogous to scheduling state under software scheduling. The priority concept, often used in software scheduling, has its analog in the priority mechanism implemented by the interrupt controller. The priority of hardware-scheduled tasks relative to software-scheduled tasks is controlled by two factors:

- The CPU's interrupt-enable flag (IF), and the instruction's CLI (which clears IF) and STI (which sets IF)
- The 8259A Programmable Interrupt Controller, an LSI component that allows selective masking of interrupts so that software can prevent some external interrupts.

When IF is set, all hardware-scheduled tasks whose interrupts are not masked out have higher priority than all software-scheduled tasks. When IF is reset, all hardware-scheduled tasks have lower priority than the currently executing task. Only the operating system (CPL <= IOPL) has the right to execute

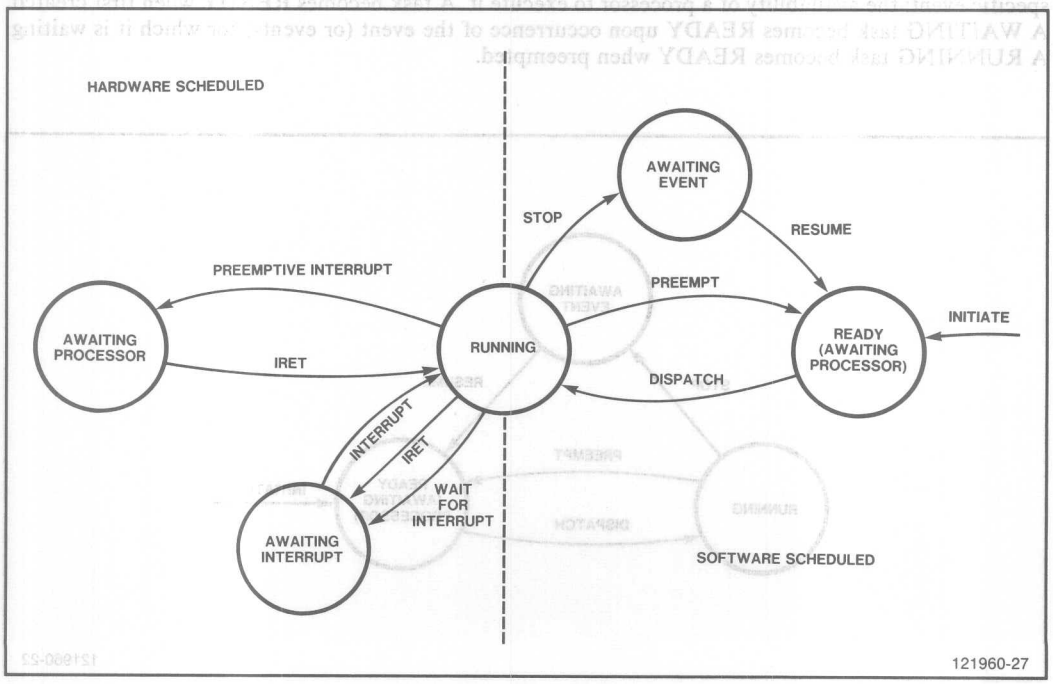


Figure 4-3. Expanded Scheduling State Transition Diagram

CLI and STI instructions due to the significant effect that priority setting has on correct, overall system operation.

The ability for an interrupt handler to be an independent task not only protects the handler from the rest of the system but also permits greater flexibility in the kinds of functions an interrupt handler can perform. An interrupt handler that is a task can use operating system functions that might change its scheduling state. For example, if an interrupt handler requests the operating system to read a disk record, the operating system may change the interrupt handler task's scheduling state from RUNNING to WAITING while the I/O operation takes place. (Other tasks may then execute in the meantime.) If the interrupt handler were a *procedure* instead of a task, it would be difficult to identify it separately from the task that it interrupted.

The operating system must keep track of whether a task is attached to an interrupt (i.e., has a task gate in the IDT). Occurrence of an interrupt can at any time dispatch the task attached to the interrupt. This happens without intervention by the operating system; therefore, when a task calls on operating system services, the operating system cannot assume that the calling task is the same as the latest software-scheduled task.

An operating system can easily determine whether the current task is hardware or software scheduled if it associates with each task a Boolean that indicates whether the task was software-scheduled. The operating system must ensure that only one task at a time is so marked. The operating system can use the STR instruction to identify the current task. If the current task is not marked as software-scheduled, then the interrupt mechanism must have dispatched it.

By knowing whether a task is attached to an interrupt, the operating system can

- Avoid executing a task that is awaiting an interrupt. A software-scheduled task cannot respond to an interrupt. An exception occurs when an interrupt attempts to invoke a busy task.
- Avoid preempting an executing interrupt task. That task should finish before software schedules another task.
- Decide what action to take when an interrupt-dispatched task calls on operating-system scheduling services. Such action might be to mask off the interrupt or to place a gate for a counting task in the IDT to mark lost interrupts.

When the operating system changes a task from hardware scheduling to software scheduling, it must update the chain of tasks that threads through TSSs. Every hardware scheduled task has a link in its TSS to the TSS of the interrupted task. If the system's interrupt structure permits nesting of interrupts, then the chain of interrupted tasks may be arbitrarily long. To change a task to software-scheduled mode, the operating system must take these actions (as figure 4-4 illustrates):

- Reset the nested-task flag (NT) of the current task.
- Nullify the back-link field in the current TSS (as insurance in case it is ever used).
- Dispatch the prior task on the back-link chain.

Changing Scheduling State

In terms of the state model of scheduling, the operating system's job is to effect transitions between scheduling states according to the expressed needs of individual tasks and of the application as a whole.

In many cases, applications drive the scheduling activities of the operating system. Applications express their scheduling needs by calls to operating system functions that indirectly relate to scheduling. For

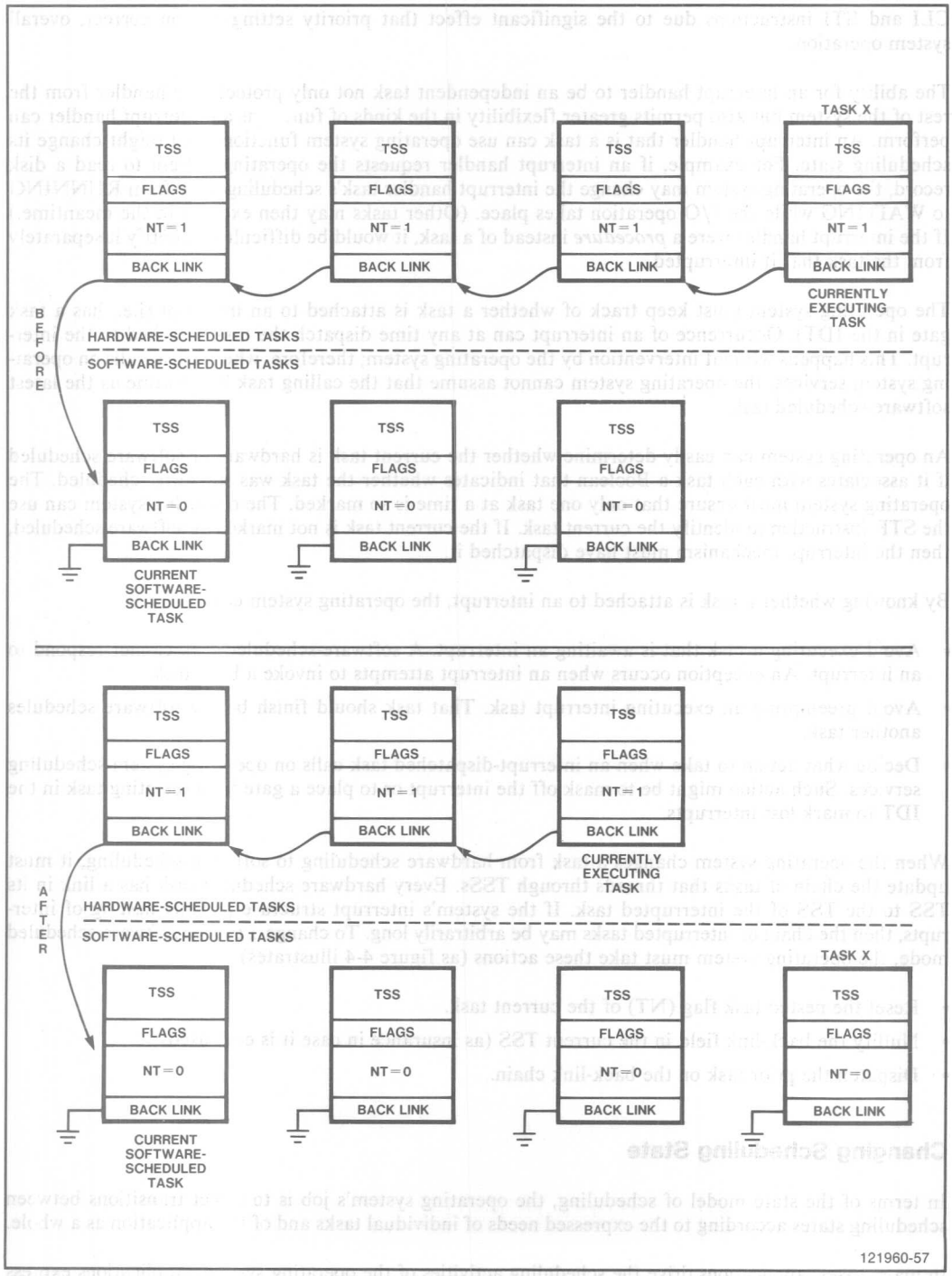


Figure 4-4. Changing Scheduling Mode

example, when an application calls the operating system to request receipt of a message from another task, the operating system determines whether a message is waiting for delivery. If no message is waiting, then the operating system must switch the task from RUNNING state to WAITING state. Later when another task calls the operating system to send a message to the waiting task, the operating system must change the task from WAITING state to READY state. In cases such as these, the operating system plays a bookkeeping role, simply keeping track of which tasks are waiting and associating events with the correct waiting tasks.

The operating system plays a much more significant role, however, when it determines which of the ready tasks to dispatch (the transition from READY state to RUNNING state) or when to preempt the task that is executing (the transition from RUNNING state to READY state). These decisions affect the overall performance of the system, both throughput and response to external events.

POLICIES AND MECHANISMS

Because of the difficulty of establishing an effective policy for dispatching and preemption decisions, it is desirable to clearly separate *mechanisms* from *policies*. The only control an operating system can exercise over tasks is deciding which task to execute and how long to let it run before changing to another task. The scheduling mechanisms must exert such control in a manner that the policies can adjust. For example, to control how long a task executes, the scheduler implements a mechanism for preempting the task after a certain time period has elapsed. The mechanism consists of an interval timer (such as Intel's 8254 Programmable Interval Timer) that interrupts the executing task periodically so that the operating system can determine whether the task has yet exceeded the time-slice allocated to it. The length of the time-slice is a variable that the policy layer can control. The policy layer sets the length of the time-slice to reflect the importance of the task.

The separation of policy and mechanism applies as well to deciding which task to execute next. For example, the scheduling mechanism associates a priority number with each task. When changing tasks it always chooses the task with highest priority. The priority, however, is a variable, and the policy layer determines its value.

For the policy layer to make reasonable decisions about scheduling, the mechanism layer may need to collect statistics about the run-time behavior of tasks, for example:

- Elapsed time in the system
- Total of actual time serviced by processor
- Running average of actual length of time-slice

(Note that interrupt-scheduled tasks subtract from the time allocated to software-scheduled tasks.)

The privilege levels of the iAPX 286 architecture can support the separation between mechanisms and policies. The mechanisms belong to the kernel of the operating system, and as such they should be well tested, highly reliable, and not subject to frequent change; in other words, they are good candidates for PL 0. Policies, on the other hand, are subject to frequent change and, as a result, are less reliable. Running scheduling policies at PL 1 ensures that errors cannot corrupt kernel procedures.

SCHEDULING POLICIES

The actual implementation of scheduling policies depends on the needs of the application and the behavior of the tasks in the system. Consider a simple policy that

- Gives all tasks equal priority
- Allocates the processor once to each task in turn
- Allocates the same maximum time-slot to each task

Even this seemingly "fair" policy actually favors certain tasks over others. A task that frequently relinquishes the processor voluntarily (because, for example, it frequently has to wait for I/O) rarely uses the full time-slot. A "processor-bound" task (for example, a computational task that uses many instructions to accomplish its purpose but rarely does I/O), on the other hand, almost always uses the full time-slot, forcing the operating system to preempt it. Over a period of many time-slots, processor-bound tasks will receive much more attention from the processor than I/O-bound tasks. Whether this situation is desirable depends on the roles of the tasks in the application.

Attempting to discriminate against processor-bound tasks by introducing a priority mechanism can result in different problems. Suppose all I/O-bound tasks have higher priority than all processor-bound tasks. At the end of a time-slice or when a task voluntarily gives up the processor, the scheduler switches to one of the higher-priority tasks if one is ready; if none is ready, it switches to one of the lower-priority tasks. The problem occurs when there is a such a number of I/O-bound tasks that at least one of them is always ready. In this case, the lower-priority, processor-bound tasks never execute.

Many more scheduling policies than those outlined here are possible. The examples given merely illustrate how important it is that the characteristics of the tasks in the system be known and that the policies match those characteristics. Refer to Coffman and Denning (see "External Literature" in the Preface) for a survey of these and other policies.

Structuring Task Information

If your operating system is to manipulate tasks efficiently, you must structure task-related information so that the operating system can get the information it needs as quickly as the application requires. The processor implements part of this structure through interlocking links in the GDT, LDT, and TSS. In addition to this structure, the operating system must deal with additional state information, which might include

- The data-segment alias to the task's LDT
- The data-segment alias to the task's TSS
- Scheduling state
- Scheduling parameters (for example, time-slice, priority)
- Scheduling statistics (for example, total processor time used, average time-slice used, expected running time)
- Links for queues of waiting and ready tasks

This additional state information is referred to here as the task database (TDB). There are two common modes of access to the task database:

1. From within the current task to information about the current task. Most operating system services use this mode of access, including the scheduler's time-slice interrupt procedure.
2. From within the current task to information about other tasks. The scheduler uses this mode of access to find the next task to execute.

ACCESS MODE 1

Access mode 1 can be efficiently implemented via the GDT. All descriptors for the key segments relating to one task reside in adjacent GDT slots. If the operating system can locate one of these descriptors (as by using the STR instruction to obtain a selector to the current TSS), then it can locate any of the others by a simple addition or subtraction. Figure 4-5 suggests one possible way of organizing the TDB. In this example the TDB is stored in a data structure called the task information block (TIB).

In large systems you may need to minimize the number of GDT slots used for task information, so as not to unduly limit either the number of tasks or the number of other descriptors that GDT can hold.

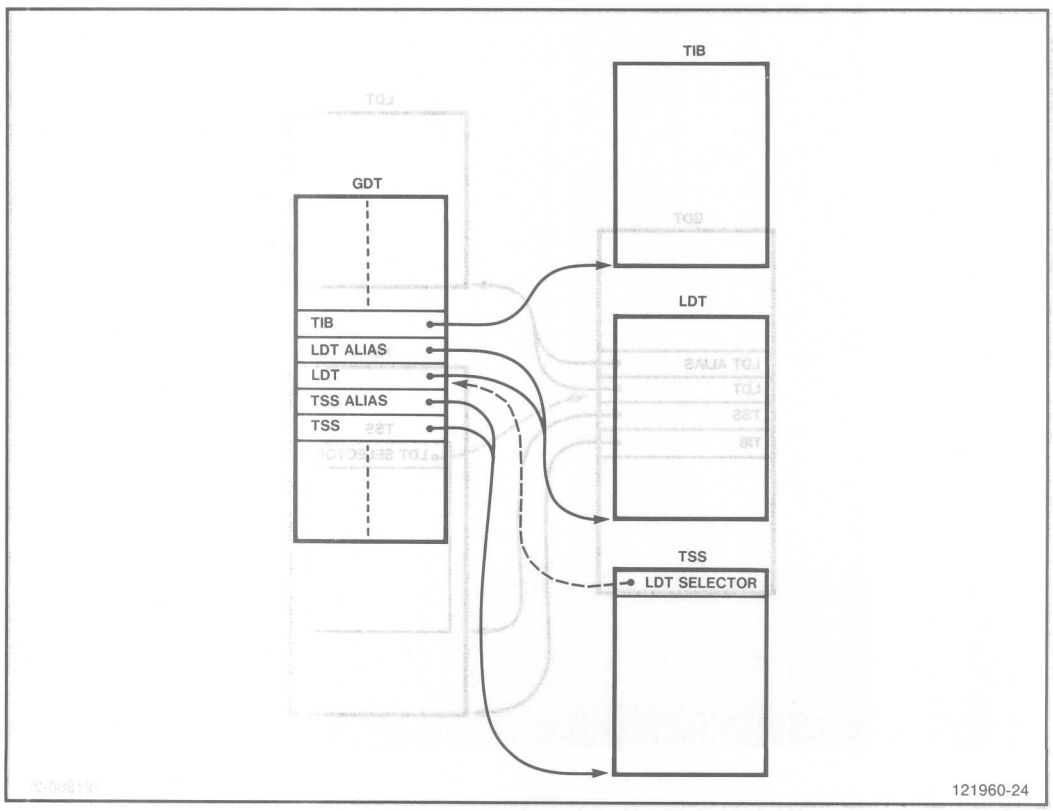


Figure 4-5. Task Information Structure A

The example in figure 4-5 illustrates the general case in which all the pertinent information is in separate segments. This case uses five GDT slots. You can free one GDT slot by including the TSS within the TIB. The TIB descriptor can serve as the data-segment alias for the TSS. Figure 4-6 illustrates such a configuration.

Speed of access to the TDB is critical in some applications. Figure 4-7 shows another configuration of task information that helps improve access speed. Here the task's stack segment for PL 0 contains both the TSS and the TIB. The advantage of this approach is that the TIB and the TSS can be addressed relative to the base address that the processor loads into SS when transferring control to a PL-0 operating-system procedure. This eliminates the need for loading the DS or ES register to access the current task's TSS or TIB and also frees a segment register for other use. In such a case, the SP portion of the TSS initial stack values for PL 0 is set to an offset beyond the TIB. In many applications it is still desirable for the GDT to hold a descriptor for the TIB so as to facilitate access to TIBs for tasks other than the current one.

ACCESS MODE 2

When the scheduler is dispatching a different task, it needs quick access to the queues of waiting and ready tasks. If the links that implement these queues thread through the many segments that contain TIBs, the time needed to search and update the queues is extended by the time needed to load a

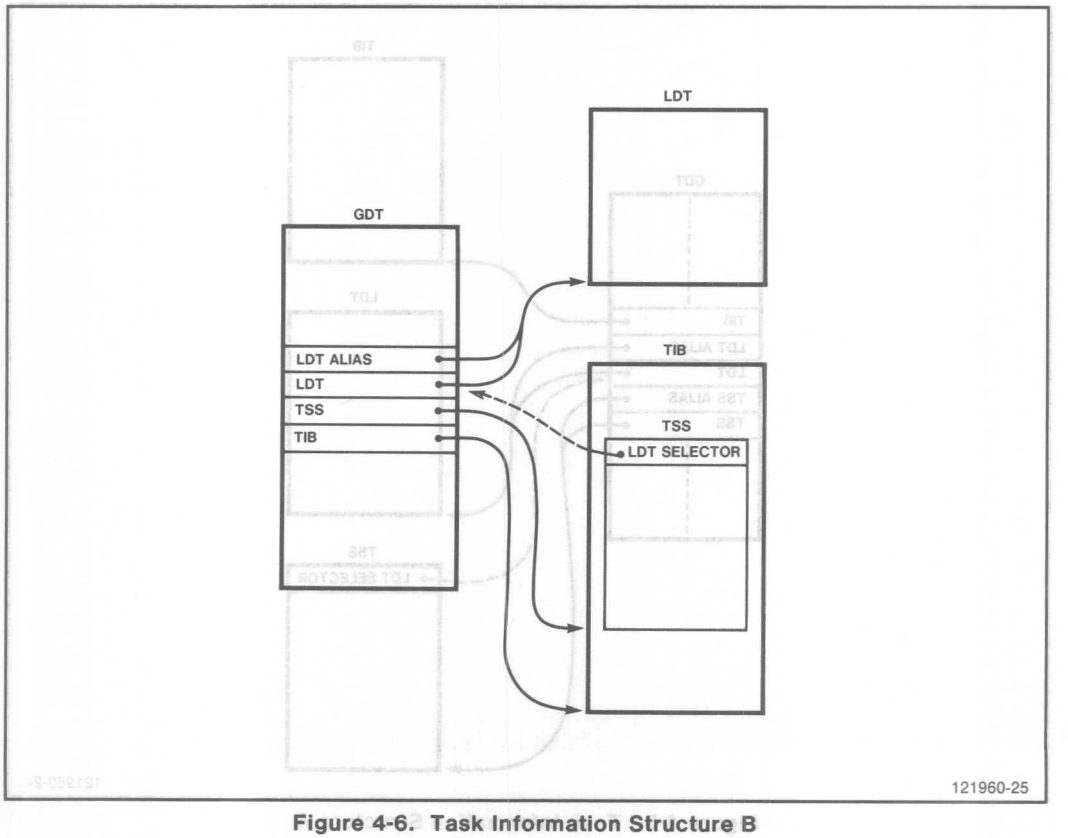


Figure 4-6. Task Information Structure B

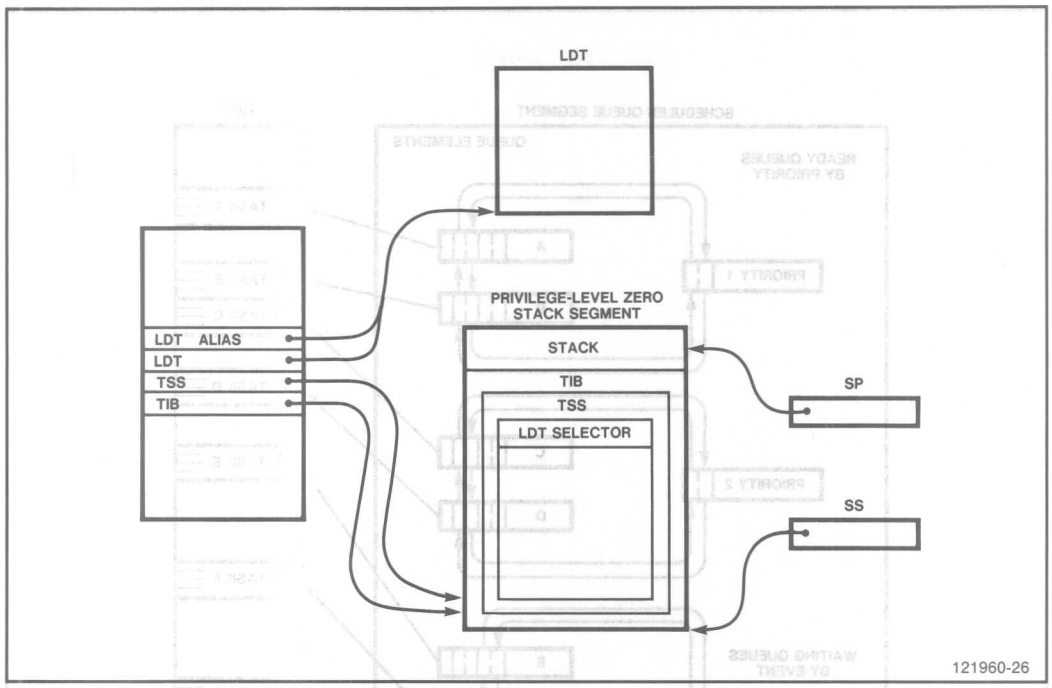


Figure 4-7. Task Information Structure C

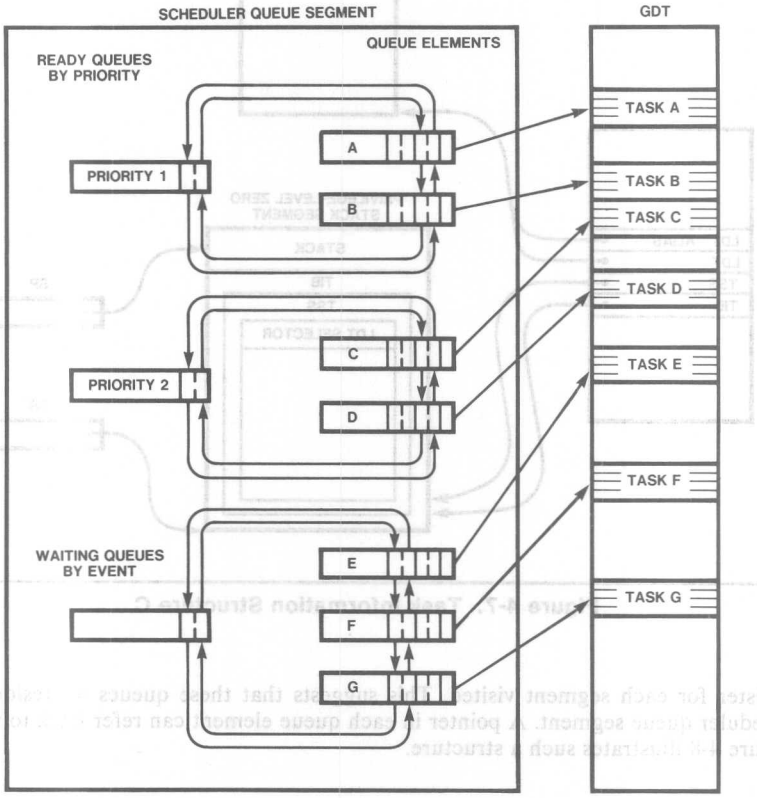
segment register for each segment visited. This suggests that these queues all reside together in a separate scheduler queue segment. A pointer in each queue element can refer back to the corresponding task. Figure 4-8 illustrates such a structure.

EXAMPLE OF A DISPATCHER

The process of changing a READY task to RUNNING state is known as "dispatching." Figure 4-9 shows a simple dispatching procedure. This procedure executes at PL 0 in the task that calls it. No call gate is provided, so only other operating system procedures can call DISPATCHER; for example, a procedure that switches the calling task to WAITING state, or a timer interrupt procedure that determines when to preempt the task that it interrupts.

The DISPATCHER procedure is coded in ASM286 instead of PL/M-286 because it is more convenient to code an indirect, intertask JMP instruction in ASM286. DISPATCHER makes two assumptions about the rest of the operating system that justify the use of an intertask JMP instead of an intertask CALL:

- A task can call operating system procedures that change it from RUNNING state, so a task does not need to execute an IRET for that purpose.
- The operating system procedure that calls DISPATCHER may set the TSS back link so as to intercept an IRET if the task executes one. (When the operating system dispatches a task, it does not make sense for an IRET to return to the previously executing task. That task may, for example, be waiting for an event and not be ready to execute.)



121960-52

Figure 4-8. Scheduler Queue Segment

121960-001

```
iAPX286 MACRO ASSEMBLER    960-503                                    04/22/83  PAGE    1

SERIES-III iAPX286 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE DISPATCHER
OBJECT MODULE PLACED IN :F5:DISP.OBJ
ASSEMBLER INVOKED BY:  ASM286.86 :F5:DISP.ASM

LOC  OBJ                  LINE     SOURCE

                             1 +1  $          TITLE ('960-503')
                             2
                             3
                             4                NAME    DISPATCHER
                             5                EXTRN   DEQUEUE_READY: FAR
                             6                PUBLIC  DISPATCHER
----                         7     STACK      STACKSEG 4
                             8
 -0004[]                     9     TSS_PTR    EQU     DWORD PTR [BP-4]
 -0002[]                    10     TSS_SEL    EQU     WORD PTR [BP-2]
 -0004[]                    11     TSS_OFFSET EQU     WORD PTR [BP-4]
                            12
----                        13     NUCLEUS_CODE       SEGMENT ER PUBLIC
                            14
0000                        15     DISPATCHER PROC    NEAR
                            16
0000 C8040000               17                ENTER   4,0             ; WE'LL FORM POINTER ON STACK
0004 FA                     18                CLI
0005 9A0000----      E      19                CALL    DEQUEUE_READY   ; RETURNS SELECTOR IN AX TO TSS
000A 3D0000                 20                CMP     AX,0            ; SAME TASK?
000D 740B                   21                JE      D_EXIT          ; JUST RETURN
                            22
000F 8946FE                 23                MOV     TSS_SEL, AX     ; FORM POINTER
0012 C746FC0000             24                MOV     TSS_OFFSET, 0   ; NOT USED ANYWAY
0017 FF6EFC                 25                JMP     TSS_PTR         ; TASK SWITCH
                            26
001A                        27           D_EXIT:
001A FB                     28                STI       ; WHEN THIS TASK EVENTUALLY REGAINS CONTROL,
001B C9                     29                LEAVE     ; IT RESUMES EXECUTING HERE, SINCE THE OFFSET
001C C3                     30                RET       ; OF THIS INSTRUCTION WAS THE LATEST VALUE IN
                            31                          ; THE IP REGISTER FOR THIS TASK
                            32
                            33     DISPATCHER ENDP
                            34
----                        35     NUCLEUS_CODE       ENDS
*** WARNING #160, LINE #35, SEGMENT CONTAINS PRIVILEGED INSTRUCTIONS
                            36                END

ASSEMBLY COMPLETE,   1 WARNING,   NO ERRORS
```

Figure 4-9. Dispatcher Example

Figure 4-9. Dispatcher Example

Data Sharing, Aliasing, and Synchronization

5

and Synchronization

CHAPTER 5
DATA SHARING, ALIASING, AND SYNCHRONIZATION

Even the simplest multitasking applications present a need for sharing data among tasks. In fact, examples presented in earlier chapters of this book have used data sharing, as in the example procedures in Chapter 3 that manipulate the free space list. For simplicity, these examples have avoided many of the implications of data sharing. However, sharing data among tasks is a complex activity that offers many opportunities for one task to cause another to fail. Therefore, for the protection of the system as a whole, the operating system must provide services that promote reliable data sharing.

DATA-SHARING TECHNIQUES

The architecture of the iAPX 286 allows segments to be shared among tasks through several mechanisms. Each mechanism has different advantages and disadvantages, and requires different degrees of support from the operating system.

Note that while the primary subject of this chapter is data sharing, these segment-sharing techniques apply as well to code segments.

Sharing via the GDT

All tasks in the system can access a segment whose descriptor resides in the GDT. This mode of sharing is especially useful for operating-system databases. Many operating-system procedures can be called from any task; they can access system data only if that data resides in segments accessible to every task.

Normally, the system designer decides in advance which descriptors are to reside in the GDT and which in LDTs, and uses the Builder to install them in the appropriate descriptor table. The only support required from the operating system is to provide synchronization for access to the shared data.

It is not always desirable to use GDT descriptors for sharing segments that only a few tasks use. A segment that has a descriptor in the GDT is accessible by all tasks, and exposing it to access from unrelated tasks may compromise the system's protection goals. Deletion of GDT descriptors may have unknown effects, because it is diffucult to control usage of GDT descriptors. GDT space may be needed for other purposes.

Sharing via Common LDT

When two (or more) tasks share most of the segments accessible to either one, then it is feasible for them to actually use that same LDT. Figure 5-1 illustrates how to share an LDT by placing the GDT selector of the LDT in the LDT field of the TSS of each task.

LDT sharing is appropriate if the sharing tasks are designed to cooperate, and if you are willing to take the risk that a failure in one task might adversely affect the other tasks that use the same LDT.

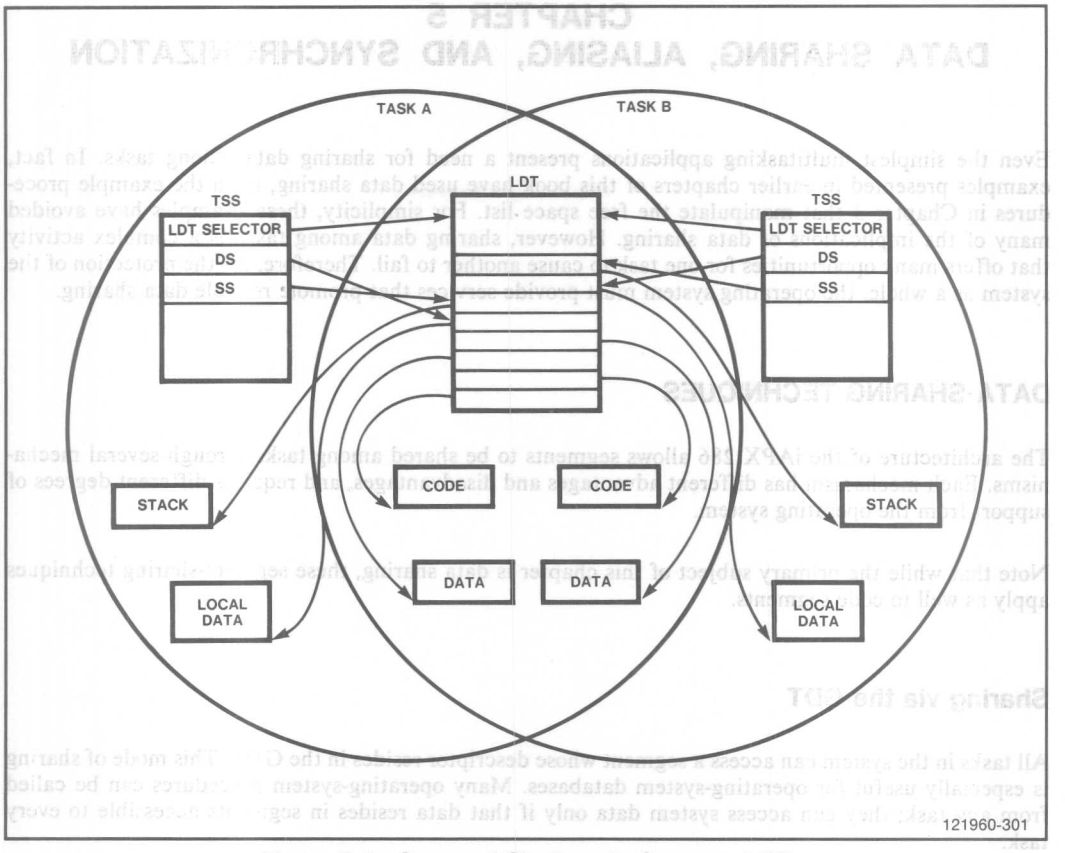


Figure 5-1. Segment Sharing via Common LDT

Sharing via Aliases

When tasks need to share simultaneously some, but not all, segments with selected other tasks, then aliasing is an appropriate method. As figure 2-15 illustrates, each task that shares a segment has a copy of the descriptor for that segment in its LDT. The term *alias* is used when there are multiple descriptors for a segment because each descriptor provides an alternate name for the segment. Not all the aliases for one segment need to be identical. Aliases may, for example, have different type or different access rights.

Descriptor manipulation must be restricted to privilege level 0 (PL 0), but procedures at any privilege level can benefit from aliasing. Therefore, the operating system must provide high-level interfaces that cause creation and deletion of aliases. The operating system makes copies of descriptors for the shared segments and installs the copies in the LDTs of the sharing tasks (or possibly at other slots in the GDT). The operating system must strictly control which tasks may receive copies of which descriptors. The presence of multiple copies of a segment's descriptor creates additional complexities for the operating system when it relocates, deletes, or otherwise modifies the segment.

Beware of the confusion that can arise when tasks share data structures that contain selectors to alias descriptors. Task A may find a selector that points to a slot in Task B's LDT. Nothing prevents Task

A from using the selector to reference its own LDT, but if it does so, the selector will likely refer to the wrong descriptor. One way to avoid this potential problem is by reserving the same slot position in the LDTs of all sharing tasks.

ALIAS MANAGEMENT

In any system that uses aliases for segment sharing, the operating system must ensure that all the aliases for a given segment remain consistent in spite of changes to the segment. The operating system may relocate a segment, transfer it to or from secondary store, or delete the segment. If allowed to use a descriptor that had not been updated to reflect any of these changes, a task would fail and might cause other tasks to fail. Therefore, an operating system must implement a means to find and update all aliases for a segment when it changes any one of the aliases.

Alias Database

Figure 5-2 shows an example method for locating aliases. This method maintains a header block for each segment that has aliases. Pointers to all the aliases for that segment are linked to the header. As long as the entire list does not span more than one segment, the link fields (FIRST, LAST, NEXT, PRIOR, and HEADER) need only contain offsets. The doubly linked list shown here aids dynamic creation and deletion of alias pointers; for static systems, a singly linked list would suffice.

In addition to the FIRST and LAST links, the list header contains that segment information that might change but that must remain consistent in all the aliases of a segment. Operating-system operations on segments demand that the segment base address and the present bit be consistent. Furthermore, any interrogation of the accessed bit for a segment demands that the accessed bits in all the aliases be ORed together. This example assumes that the operating system does not permit creation of aliases with differing limit or expansion direction.

When a task that has a selector for an alias descriptor calls on operating system functions that make changes to segment attributes, those changes must be broadcast to all other aliases for the segment. Therefore, the operating system must have a means, given any descriptor, to find the alias list that includes that descriptor. Figure 5-3 illustrates one technique for doing this. Each descriptor table has a parallel table of pointers to alias list headers. The index in a selector that locates a descriptor in the descriptor table also locates a pointer in the parallel table. A descriptor that has no alias has a null entry in the corresponding position in the parallel table. For applications in which aliases are few, you can employ a hashing algorithm to reduce the number of entries in the parallel table.

Alias Procedures

Implementation of procedures for alias management for the 80286 is a straightforward application of list processing algorithms and therefore is not illustrated here. At a minimum, the operating system should provide a CREATE_ALIAS procedure and a DELETE_ALIAS procedure.

The operating system must enforce a correspondence between the existence of descriptors for a segment and the existence of the segment itself. A segment must always have a descriptor, and an active descriptor must always point to a valid segment. A convenient way to enforce these rules is to permit only the DELETE_ALIAS procedure to call the segment FREE procedure. DELETE_ALIAS should cause deletion of a segment only when the last descriptor for the segment has been deleted.

Creating an alias is only the first step toward segment sharing. The CREATE_ALIAS procedure can only create an alias for a segment that is accessible to the task that calls CREATE_ALIAS. The next

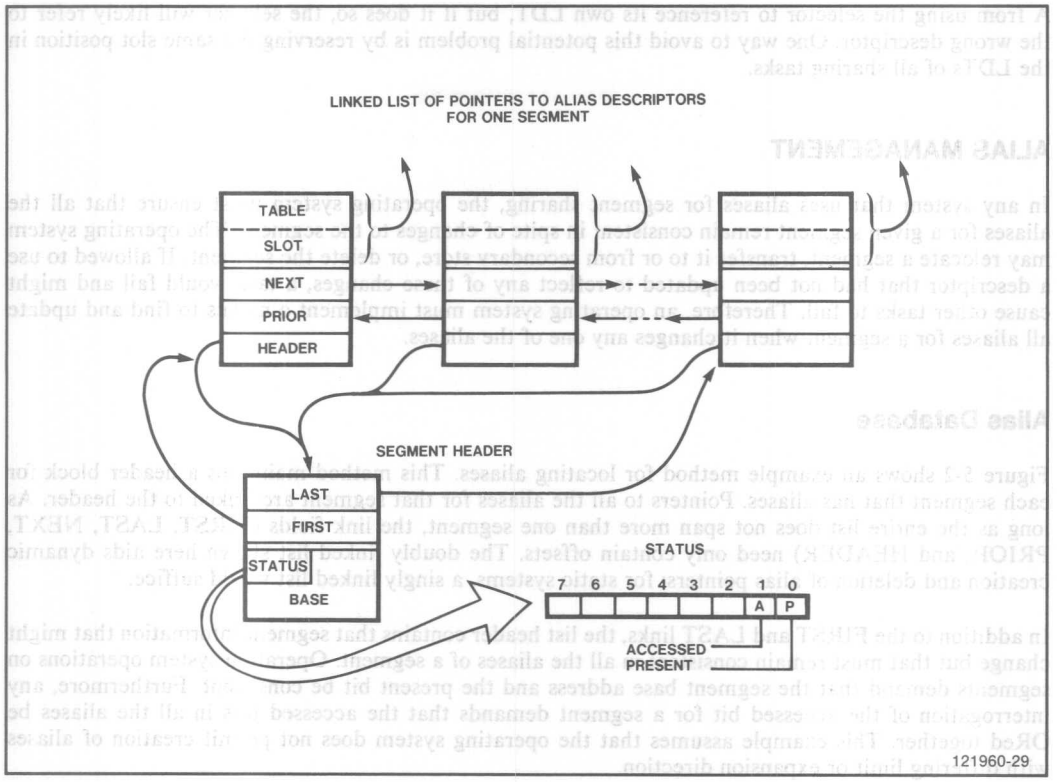


Figure 5-2. Alias List

step toward segment sharing is to pass the alias to another task. This subject is taken up in a later section of this chapter, "Message Passing."

SYNCHRONIZATION

Consider the example of a memory manager given in Chapter 3. While a memory management procedure manipulates the links that manage free space, there are instants when the data in the links is temporarily inconsistent (for example, the next-pointer in one segment has been set, but the previous-pointer in the next segment has not yet been updated). If the processor interrupts the task in which the memory-management procedure is running to run another task (or even another procedure in the same task) and if the new task (or procedure) calls the memory manager, then the the memory manager is likely to behave incorrectly.

For the protection of the system, the operating system must prevent such forms of incorrect function in its own logic and must provide synchronization operations that enable application logic to avoid such failures as well.

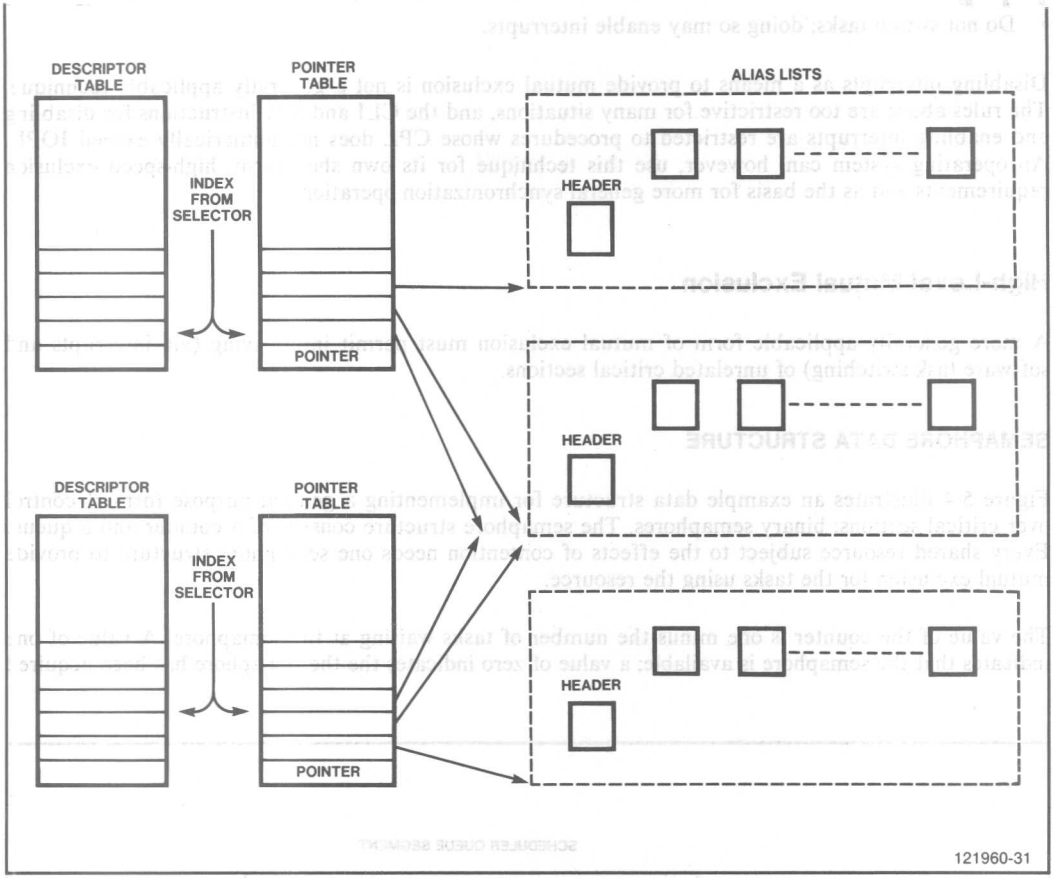


Figure 5-3. Identifying Alias List

Low-Level Mutual Exclusion

The most basic form of synchronization is control over *critical sections*. A critical section is a sequence of instructions that operates on shared resources in such a way that errors could result if another sequence of instructions operated on the same resources within the same time span. Any means that prevents any two critical sections from overlapping in time would prevent such errors. In a single-processor system, only an interrupt can cause the operations of one critical section to interleave with those of another. Disabling interrupts provides the necessary mutual exclusion.

Procedures that disable interrupts to provide mutual exclusion must adhere to certain rules:

- Determine the maximum permissible delay for servicing an interrupt, and do not code a critical section that takes more time.
- Avoid causing a fault that might keep interrupts disabled for a longer time than permissible.

- Do not nest critical sections; there is only one interrupt flag.
- Do not switch tasks; doing so may enable interrupts.

Disabling interrupts as a means to provide mutual exclusion is not a generally applicable technique. The rules above are too restrictive for many situations, and the CLI and STI instructions for disabling and enabling interrupts are restricted to procedures whose CPL does not numerically exceed IOPL. An operating system can, however, use this technique for its own short-term, high-speed exclusion requirements and as the basis for more general synchronization operations.

High-Level Mutual Exclusion

A more generally applicable form of mutual exclusion must permit interleaving (via interrupts and software task switching) of unrelated critical sections.

SEMAPHORE DATA STRUCTURE

Figure 5-4 illustrates an example data structure for implementing a general purpose form of control over critical sections: binary semaphores. The semaphore structure consists of a counter and a queue. Every shared resource subject to the effects of contention needs one semaphore structure to provide mutual exclusion for the tasks using the resource.

The value of the counter is one minus the number of tasks waiting at the semaphore. A value of one indicates that the semaphore is available; a value of zero indicates the the semaphore has been acquired

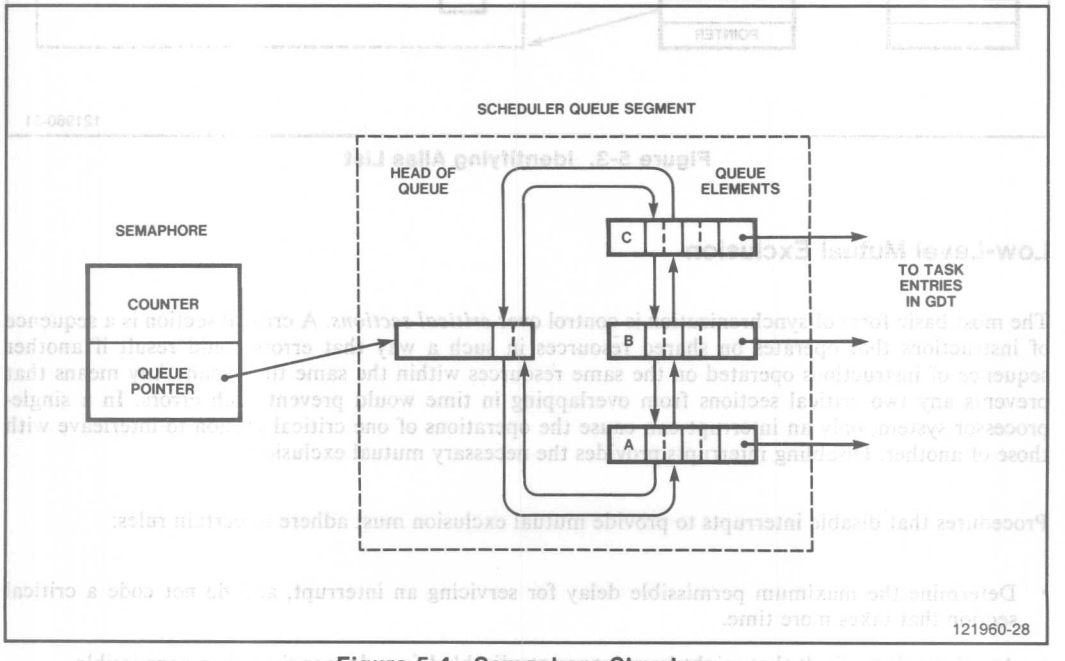


Figure 5-4. Semaphore Structure

but no other tasks are waiting for the semaphore; a negative value indicates that some tasks are queued, waiting for the semaphore to be released. The queue is circular so that the pointer to the head of the queue also identifies the tail of the queue.

Semaphore structures may be stored individually in separate segments or may be stored together in one or more segments. In the latter case the operating system must provide a means for identifying individual semaphores within a segment. Storing the semaphore counter in a segment by itself yields two important advantages:

- The processor can individually protect each semaphore.
- The selector of the descriptor for the semaphore segment serves as a convenient identifier for the semaphore.

Semaphore structures are sensitive data that must be cloistered behind the level 0 protection wall. Semaphores may reside either in the GDT or in the LDTs of the tasks that use them. The same considerations apply as for shared data segments.

SEMAPHORE-MANAGEMENT PROCEDURES

The minimum set of operating-system procedures needed to use semaphores includes

- WAIT_SEMAPHORE to acquire a semaphore if it is not already acquired
- SIGNAL_SEMAPHORE to signal departure from a critical section

Dynamic systems may also need to define semaphores dynamically with procedures such as

- CREATE_SEMAPHORE for setting up a new semaphore structure
- DELETE_SEMAPHORE to eliminate a semaphore structure that is no longer needed

The operating system's responsibilities in managing binary semaphores include

- Ensuring that no more than one task holds a semaphore
- When a task requests a semaphore that has not been signalled, placing the task in a waiting queue, and dispatching another task
- When a task signals a semaphore, awakening the next task in the waiting queue for that semaphore and placing it in the ready queue
- Preventing or being prepared to handle any deadlock that may result from using semaphores

Figure 5-5 shows simple examples of how to use a low-level synchronization technique to implement WAIT_SEMAPHORE and SIGNAL_SEMAPHORE procedures. These procedures must run at PL 0 to access the semaphore structures. With segment descriptors in the GDT and with call gates at PL 3, any procedure in the system can call these synchronization primitives.

```
PL/M-286 COMPILER     960-504                    date              PAGE   1


system-ID  PL/M-286 Vx.y COMPILATION OF MODULE SEMAPH
OBJECT MODULE PLACED IN :F1:SEMAPH.OBJ
COMPILER INVOKED BY:  PLM286.86 :F1:SEMAPH.PLM DEBUG


                  $ PAGEWIDTH(71) TITLE('960-504') INCLUDE (:F1:NUCSUB.PLM)
              =   $ NOLIST

    1             SEMAPH:  DO;

                  /*******************************************************/
                  /*                      Externals                      */

    2   1         DISPATCHER: PROCEDURE EXTERNAL;
    3   2         END DISPATCHER;

    4   1         ENQUEUE_WAIT: PROCEDURE(QUEUE_ID) EXTERNAL;
    5   2          DECLARE QUEUE_ID SELECTOR;
    6   2         END ENQUEUE_WAIT;

    7   1         DEQUEUE_WAIT: PROCEDURE(QUEUE_ID,EXCEP_P) EXTERNAL;
    8   2          DECLARE QUEUE_ID SELECTOR, EXCEP_P POINTER;
    9   2         END DEQUEUE_WAIT;

                  /*******************************************************/
                  /*             Semaphore Data Structures               */

   10   1         DECLARE SEMAPHORMAT LITERALLY
                   'FILLER (2) WORD,
                    COUNTER    WORD';

   11   1         DECLARE OK  LITERALLY '0';

                  /*******************************************************/
                  /*        Test a semaphore; wait if not set            */

   12   1         WAIT_SEMAPHORE:  PROCEDURE (SEMAPH_ID, EXCEP_P)
                                                 PUBLIC REENTRANT;

   13   2          DECLARE SEMAPH_ID   SELECTOR,
                          SEMAPH BASED SEMAPH_ID STRUCTURE(SEMAPHORMAT);

   14   2          DECLARE EXCEP_P              POINTER,
                          EXCEP BASED EXCEP_P  WORD;

   15   2          DISABLE;
   16   2          SEMAPH.COUNTER=SEMAPH.COUNTER-1;
   17   2          IF ZERO /* Test the zero flag. */
   18   2           THEN /* Semaphore was set. */ DO;
   19   3            ENABLE;
   20   3            EXCEP=OK;
   21   3            RETURN;
   22   3           END;

                   /* Semaphore is not set; this task must wait. */
   23   2          CALL ENQUEUE_WAIT (SEMAPH_ID);
```

Figure 5-5. Semaphore Example

```
PL/M-286 COMPILER     960-504                    date              PAGE   2


   24   2        ENABLE;
   25   2        CALL DISPATCHER;

   26   2      END WAIT_SEMAPHORE;

                /*******************************************************/
                /*                  Set a Semaphore                    */
   27   1      SIGNAL_SEMAPHORE:  PROCEDURE (SEMAPH_ID, EXCEP_P)
                                                    PUBLIC REENTRANT;

   28   2        DECLARE SEMAPH_ID    SELECTOR,
                        SEMAPH BASED SEMAPH_ID STRUCTURE(SEMAPHORMAT);

   29   2        DECLARE EXCEP_P                POINTER,
                        EXCEP BASED EXCEP_P  WORD;
   30   2        DISABLE;
   31   2        SEMAPH.COUNTER=SEMAPH.COUNTER+1;
   32   2        IF NOT (ZERO OR SIGN) /* Test flags. */
   33   2         THEN /* No one is waiting at this semaphore. */ DO;
   34   3           ENABLE;
   35   3           EXCEP=OK;
   36   3           RETURN;
   37   3         END;

                /* Someone is waiting at this semaphore. */
   38   2        CALL DEQUEUE_WAIT (SEMAPH_ID, @EXCEP);
   39   2        ENABLE;
   40   2        CALL DISPATCHER;
   41   2        EXCEP=OK;

   42   2      END SIGNAL_SEMAPHORE;

                /*******************************************************/

   43   1      END SEMAPH;


MODULE INFORMATION:

     CODE AREA SIZE     = 0084H    132D
     CONSTANT AREA SIZE = 0000H     0D
     VARIABLE AREA SIZE = 0000H     0D
     MAXIMUM STACK SIZE = 0010H    16D
     107 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     91KB MEMORY AVAILABLE
     4KB MEMORY USED   (4%)
     0KB DISK SPACE USED

END OF PL/M-286 COMPILATION
```

Figure 5-5. Semaphore Example (Cont'd.)

OTHER FORMS OF SYNCHRONIZATION

You may wish to implement other forms of synchronization; for example:

- Conditional variation of WAIT_SEMAPHORE that does not force a task to wait when the semaphore has not been signalled.
- Timed waiting, so that a task can be awakened if the semaphore is not signalled within a reasonable time. (This helps guard against deadlocks.)
- An extension of the semaphore concept known as a *region*. A region is similar to a semaphore except that only the task that acquires a region can release (signal) it, and a task that holds a region cannot be suspended.

MESSAGE PASSING

Message passing is a general purpose means for transferring data from the address space of one task to the address space of another, cooperating task. There are two aspects to the technique:

- Transferring data from a segment in one task's address space into a segment in the receiving task's space
- Transferring a *segment* from one task's space into another's

The first case is suitable for passing relatively small amounts of data, such as parameters, information describing events, etc. The second case has two primary applications:

- For transferring large "consumable resources" such as I/O buffers
- For transferring aliases that implement the previously described method of "sharing via aliases"

When an alias is passed as part of a message, the operating system installs the alias in a descriptor-table slot determined by the receiving task.

Message-Passing Example

Figure 5-6 shows an example data structure for implementing a simple form of message passing. This structure defines a mailbox, a queue of tasks waiting for messages from the mailbox, and a queue of undelivered messages. The system may contain one mailbox for every communication channel between tasks. For simplicity, this example assumes that the format of messages for all mailboxes is the same, consisting of a fixed-length data item and two descriptors.

If each mailbox resides in a unique segment, then these advantages result:

- Mailboxes are protected from operations on other mailboxes.
- A selector can serve as the identifier of a mailbox.

Only the sending and receiving tasks need access to a mailbox; therefore, the appropriate tables for descriptors for mailbox segments are the LDTs of each of the tasks that share a mailbox. All tasks can share a global mailbox, however, if its descriptor is in the GDT. The DPL for all mailbox segments should be zero to prevent procedures outside the operating system from interfering with message passing.

Mailboxes require at least two procedures: SEND_MESSAGE and RECEIVE_MESSAGE. Figure 5-7 shows examples of these procedures. Both procedures must run at PL 0 to access level-0 mailbox

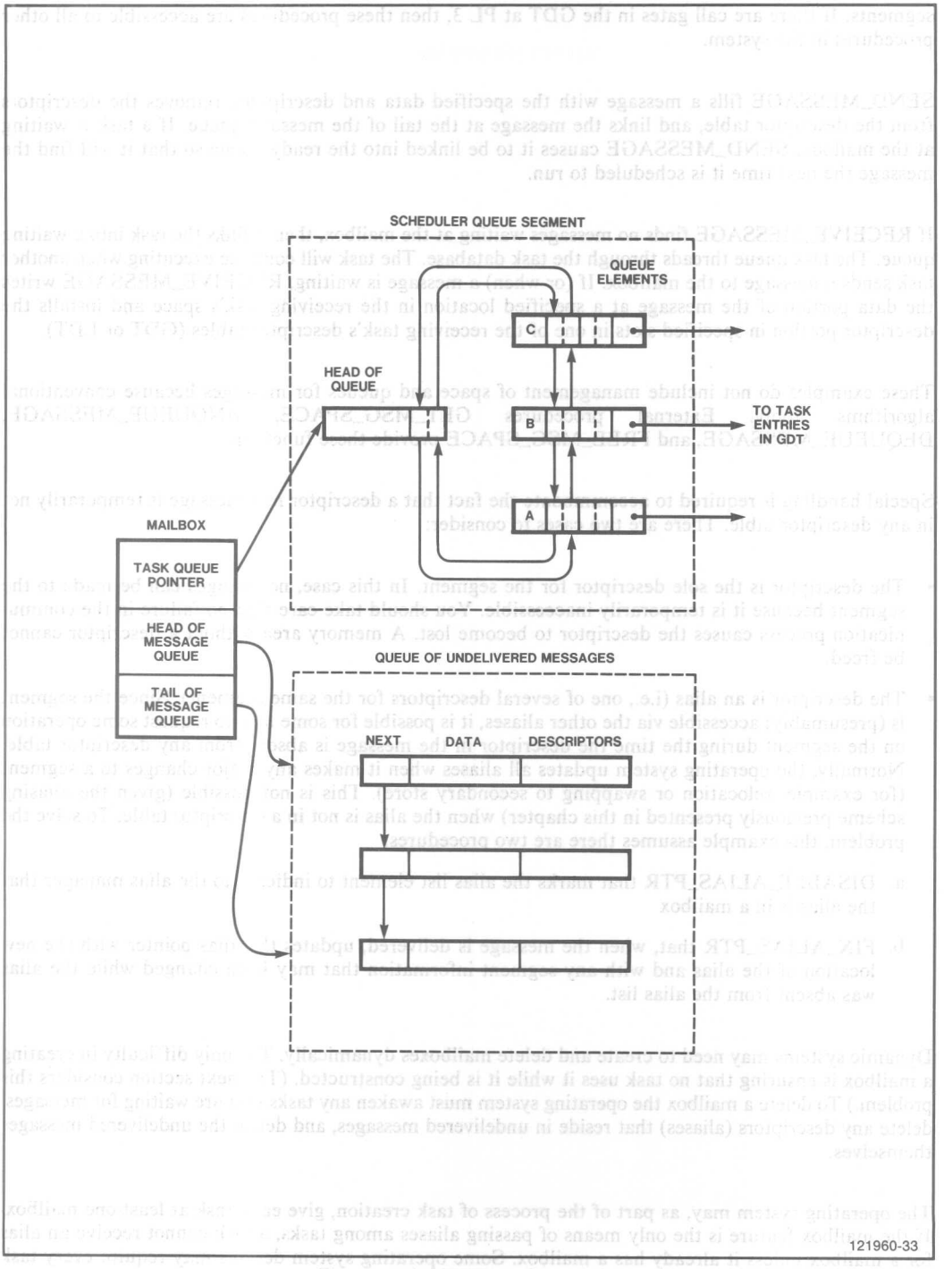


Figure 5-6. Mailbox Structure

segments. If there are call gates in the GDT at PL 3, then these procedures are accessible to all other procedures in the system.

SEND_MESSAGE fills a message with the specified data and descriptors, removes the descriptors from the descriptor table, and links the message at the tail of the message queue. If a task is waiting at the mailbox, SEND_MESSAGE causes it to be linked into the ready queue so that it will find the message the next time it is scheduled to run.

If RECEIVE_MESSAGE finds no messages waiting at the mailbox, then it links the task into a waiting queue. The task queue threads through the task database. The task will continue executing when another task sends a message to the mailbox. If (or when) a message is waiting, RECEIVE_MESSAGE writes the data portion of the message at a specified location in the receiving task's space and installs the descriptor portion in specified slots in one of the receiving task's descriptor tables (GDT or LDT).

These examples do not include management of space and queues for messages because conventional algorithms apply. External procedures GET_MSG_SPACE, ENQUEUE_MESSAGE, DEQUEUE_MESSAGE, and FREE_MSG_SPACE provide these functions.

Special handling is required to accommodate the fact that a descriptor in a message is temporarily not in any descriptor table. There are two cases to consider:

- The descriptor is the sole descriptor for the segment. In this case, no changes can be made to the segment because it is temporarily inaccessible. You should take care that no failure in the communication process causes the descriptor to become lost. A memory area without a descriptor cannot be freed.
- The descriptor is an alias (i.e., one of several descriptors for the same segment). Since the segment is (presumably) accessible via the other aliases, it is possible for some task to request some operation on the segment during the time the descriptor in the message is absent from any descriptor table. Normally, the operating system updates all aliases when it makes any major changes to a segment (for example, relocation or swapping to secondary store). This is not possible (given the aliasing scheme previously presented in this chapter) when the alias is not in a descriptor table. To solve the problem, this example assumes there are two procedures:
  a. DISABLE_ALIAS_PTR that marks the alias list element to indicate to the alias manager that the alias is in a mailbox
  b. FIX_ALIAS_PTR that, when the message is delivered, updates the alias pointer with the new location of the alias and with any segment information that may have changed while the alias was absent from the alias list.

Dynamic systems may need to create and delete mailboxes dynamically. The only difficulty in creating a mailbox is ensuring that no task uses it while it is being constructed. (The next section considers this problem.) To delete a mailbox the operating system must awaken any tasks that are waiting for messages, delete any descriptors (aliases) that reside in undelivered messages, and delete the undelivered messages themselves.

The operating system may, as part of the process of task creation, give each task at least one mailbox. If the mailbox feature is the only means of passing aliases among tasks, a task cannot receive an alias for a mailbox unless it already has a mailbox. Some operating system designs may require every task to have a mailbox for such purposes as receiving memory segments from the memory-allocation module.

```
PL/M-286 COMPILER    960-508                    date              PAGE   1


system-ID  PL/M-286 Vx.y COMPILATION OF MODULE MAILBOX
OBJECT MODULE PLACED IN :F1:MBOX.OBJ
COMPILER INVOKED BY:  PLM286.86 :F1:MBOX.PLM DEBUG


           $ PAGEWIDTH(71) TITLE('960-508') INCLUDE (:F1:NUCSUB.PLM)
       =   $ NOLIST

 1         MAILBOX:  DO;

           /*************************************************/
           /*                  Definitions                  */

 2   1     DECLARE FAILED  LITERALLY '8000H',
                 OK      LITERALLY '0';

           /*************************************************/
           /*                  Externals                    */

 3   1     NULLIFY: PROCEDURE (SLOT) EXTERNAL;
 4   2      DECLARE SLOT   SELECTOR;
 5   2     END NULLIFY;

 6   1     STORE_DESCR: PROCEDURE(SLOT,PTR) EXTERNAL;
 7   2      DECLARE SLOT SELECTOR,
                    PTR  POINTER;
 8   2     END STORE_DESCR;

 9   1     LOAD_DESCR: PROCEDURE(PTR,SLOT) EXTERNAL;
10   2      DECLARE PTR  POINTER,
                     SLOT SELECTOR;
11   2     END LOAD_DESCR;

12   1     DISPATCHER: PROCEDURE EXTERNAL;
13   2     END DISPATCHER;

14   1     ENQUEUE_WAIT: PROCEDURE(QUEUE_ID) EXTERNAL;
15   2      DECLARE QUEUE_ID SELECTOR;
16   2     END ENQUEUE_WAIT;

17   1     DEQUEUE_WAIT: PROCEDURE(QUEUE_ID, EXCEP_P) EXTERNAL;
18   2      DECLARE QUEUE_ID SELECTOR, EXCEP_P  POINTER;
19   2     END DEQUEUE_WAIT;

20   1     DISABLE_ALIAS_PTR: PROCEDURE(SLOT) EXTERNAL;
21   2      DECLARE SLOT SELECTOR;
22   2     END DISABLE_ALIAS_PTR;

23   1     FIX_ALIAS_PTR: PROCEDURE(ALIAS_LIST_ID) EXTERNAL;
24   2      DECLARE ALIAS_LIST_ID POINTER;
25   2     END FIX_ALIAS_PTR;

26   1     GET_MSG_SPACE: PROCEDURE(BOX_ID,MSG_P_P,EXCEP_P)EXTERNAL;
27   2      DECLARE BOX_ID  SELECTOR, (MSG_P_P, EXCEP_P) POINTER;
28   2     END GET_MSG_SPACE;
```

Figure 5-7. Example of Mailbox Procedures

```
PL/M-286 COMPILER    960-508                  date              PAGE   2


  29   1       FREE_MSG_SPACE: PROCEDURE(BOX_ID,MSG_PTR) EXTERNAL;
  30   2        DECLARE BOX_ID   SELECTOR,
                       MSG_PTR  POINTER;
  31   2       END FREE_MSG_SPACE;

  32   1       ENQUEUE_MESSAGE: PROCEDURE(BOX_ID, MSG_PTR) EXTERNAL;
  33   2        DECLARE BOX_ID   SELECTOR,
                       MSG_PTR  POINTER;
  34   2       END ENQUEUE_MESSAGE;

  35   1       DEQUEUE_MESSAGE: PROCEDURE (BOX_ID, MSG_P_P, EXCEP_P)
                                                          EXTERNAL;
  36   2        DECLARE BOX_ID SELECTOR, (MSG_P_P, EXCEP_P) POINTER;
  37   2       END DEQUEUE_MESSAGE;

                /*******************************************************/
                /*                 Mailbox Data Structures             */

  38   1       DECLARE MDATA_SIZE          LITERALLY '46';

  39   1       DECLARE MESSAGE_FORMAT      LITERALLY
                       'MDATA (MDATA_SIZE) BYTE,
                        DESCR1 (4)         WORD,
                        DESCR2 (4)         WORD';

                /*******************************************************/
                /*             Send Message via Mailbox                */

  40   1       SEND_MESSAGE: PROCEDURE(BOX_ID, MDATA_PTR, SLOT1, SLOT2,
                                       EXCEP_P)      PUBLIC REENTRANT;

  41   2        DECLARE BOX_ID          SELECTOR,
                       MDATA_PTR       POINTER,
                       (SLOT1, SLOT2) SELECTOR;

  42   2        DECLARE EXCEP_P                 POINTER,
                       EXCEP  BASED EXCEP_P    WORD;

  43   2        DECLARE MSG_PTR   POINTER,
                       MESSAGE BASED MSG_PTR STRUCTURE
                         (MESSAGE_FORMAT);

  44   2        CALL GET_MSG_SPACE(BOX_ID, @MSG_PTR, @EXCEP);
  45   2        IF EXCEP=FAILED THEN /* the box is full of messages */
  46   2         DO;
  47   3          CALL DISPATCHER;
  48   3          RETURN;
  49   3         END;
                 /* The next statement will cause an exception if the
                    segment containing the data is not present.
                    Therefore interrupts are enabled.          */
  50   2        CALL MOVB (MDATA_PTR,@MESSAGE.MDATA,MDATA_SIZE);
  51   2        IF SLOT1=SELECTOR$OF(NIL)
  52   2         THEN MESSAGE.DESCR1(2)=0; /* Mark as null */
  53   2         ELSE DO;
  54   3          CALL STORE_DESCR(SLOT1,@MESSAGE.DESCR1);
  55   3          CALL DISABLE_ALIAS_PTR(SLOT1);
```

Figure 5-7. Example of Mailbox Procedures (Cont'd.)

```
PL/M-286 COMPILER     960-508                    date                PAGE   3


  56   3          CALL NULLIFY(SLOT1);
  57   3         END;
  58   2        IF SLOT2=SELECTOR$OF(NIL)
  59   2         THEN MESSAGE.DESCR2(2)=0; /* Mark as null */
  60   2         ELSE DO;
  61   3          CALL STORE_DESCR(SLOT2,@MESSAGE.DESCR2);
  62   3          CALL DISABLE_ALIAS_PTR(SLOT2);
  63   3          CALL NULLIFY(SLOT2);
  64   3         END;
  65   2        DISABLE;
  66   2        CALL ENQUEUE_MESSAGE(BOX_ID, MSG_PTR);
  67   2        CALL DEQUEUE_WAIT(BOX_ID,@EXCEP);
  68   2        ENABLE;
  69   2        CALL DISPATCHER;
  70   2        RETURN;

  71   2      END SEND_MESSAGE;

                /*********************************************************/
                /*            Receive Message from Mailbox               */

  72   1      RECEIVE_MESSAGE: PROCEDURE (BOX_ID, MDATA_PTR, SLOT1,
                                 SLOT2, EXCEP_P) PUBLIC REENTRANT;

  73   2       DECLARE BOX_ID            SELECTOR,
                       MDATA_PTR         POINTER,
                       (SLOT1, SLOT2)    SELECTOR;

  74   2       DECLARE EXCEP_P               POINTER,
                       EXCEP BASED EXCEP_P   WORD;

  75   2       DECLARE MSG_PTR   POINTER,
                       MESSAGE BASED MSG_PTR STRUCTURE
                        (MESSAGE_FORMAT);

  76   2      CHECK_MAIL:
               DISABLE;
  77   2       CALL DEQUEUE_MESSAGE (BOX_ID, @MSG_PTR, @EXCEP);
  78   2       IF EXCEP=FAILED THEN /* No mail today */
  79   2        DO;
  80   3         CALL ENQUEUE_WAIT (BOX_ID);
  81   3         ENABLE;
  82   3         CALL DISPATCHER;
  83   3         GOTO CHECK_MAIL;
  84   3        END;
  85   2       ENABLE;

                /* Next statement may cause exception. */
  86   2       CALL MOVB(@MESSAGE.MDATA, MDATA_PTR, MDATA_SIZE);
  87   2       IF MESSAGE.DESCR1(2)<>0 /* Test for null descriptor */
  88   2        THEN DO;
  89   3         CALL LOAD_DESCR(@MESSAGE.DESCR1, SLOT1);
  90   3         CALL FIX_ALIAS_PTR(@MESSAGE.DESCR1);
  91   3        END;
  92   2       IF MESSAGE.DESCR2(2)<>0 /* Test for null descriptor */
  93   2        THEN DO;
  94   3         CALL LOAD_DESCR(@MESSAGE.DESCR2, SLOT2);
```

Figure 5-7. Example of Mailbox Procedures (Cont'd.)

```
PL/M-286 COMPILER     960-508                    date              PAGE   4


  95    3          CALL FIX_ALIAS_PTR(@MESSAGE.DESCR2);
  96    3        END;
  97    2       CALL FREE_MSG_SPACE(BOX_ID,@MESSAGE);
  98    2       EXCEP=OK;

  99    2      END RECEIVE_MESSAGE;

               /*******************************************************/

 100    1      END MAILBOX;


MODULE INFORMATION:

     CODE AREA SIZE      = 016EH     366D
     CONSTANT AREA SIZE  = 0000H       0D
     VARIABLE AREA SIZE  = 0000H       0D
     MAXIMUM STACK SIZE  = 0020H      32D
     193 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     91KB MEMORY AVAILABLE
     6KB MEMORY USED    (6%)
     0KB DISK SPACE USED

END OF PL/M-286 COMPILATION
```

Figure 5-7. Example of Mailbox Procedures (Cont'd.)

Variations on the Mailbox Theme

Some of the features appropriate for semaphores are also appropriate for mailboxes, namely:

- Conditional receive
- Timed wait

For applications in which speed is not the overriding goal, mailboxes can substitute for semaphores. A mailbox is analogous to a semaphore, with the counter of a semaphore corresponding to the number of messages waiting at a mailbox. A mailbox with null messages is identical in function to a semaphore.

Most applications that use mailboxes require different message formats (different data size, different number of descriptors) for different communication channels. With dynamically created mailboxes, message format may be defined by parameters to the creation procedure, encoded in the mailbox, and interpreted by the SEND_MESSAGE and RECEIVE_MESSAGE procedures.

Systems that implement "pipes" as a form of intertask communication can call mailbox procedures from the device drivers for the pipes.

It is possible to use mailboxes as the sole means of interface between applications and operating system. For example, the operating system has one mailbox through which it receives service requests from all applications; each task has a mailbox through which it receives responses from the operating system. To get more memory, a task would send a memory request message to the operating system; the operating system would return a message containing an alias for the allocated memory segment and install the alias in the task's LDT. The advantage is simplicity. Only two global gates are needed: one for SEND_MESSAGE and one for RECEIVE_MESSAGE. A task can wait for only one purpose: for a message from a mailbox. The disadvantage is inefficiency. Any implementation of mailboxes is bound to be less efficient than the interlevel CALL instruction normally used to communicate with an operating system.

All of these forms of message-passing can use the primitive synchronization and descriptor-manipulation techniques illustrated in this section.B

Signals and Interrupts 6

6 Signals and Interrupts

CHAPTER 6
SIGNALS AND INTERRUPTS

Interrupts are a mechanism long used in single-task microprocessors for reacting quickly to external events. In the multitasking architecture of the iAPX 286, each task may have the same needs for information about external events as the one task in a single-task system. In a multitasking system, several generalizations of interrupts are useful:

- Some tasks may do nothing but service a specific external event. While the task waits for an event to occur, the processor can service other tasks.
- Information about an external event must be routed to the correct task.
- Events external to a task may include events occurring in other tasks as well as events external to the processor.
- The ability to selectively ignore events must then extend to those events occurring in other tasks.
- Each task can benefit from a vector table to automatically route information about events to the correct handler procedures within the task.
- Scheduling of tasks that service events must be coordinated with the software scheduler, while retaining the ability to respond rapidly to events.

The 80286 implements some of these generalizations of interrupts onto the multitasking environment, but the operating system has responsibility for others. Not all are relevant to every application of the 80286.

INTERRUPT FEATURES OF THE iAPX 286 ARCHITECTURE

The iAPX 286 architecture includes a number of features that work together to enable efficient response to events.

Vectoring

The processor associates each event with an identifying number in the range 0–255. The processor recognizes three classes of events:

- External. Events occurring outside the 80286 processor's environment are communicated to the processor via the INTR or NMI (non-maskable interrupt) pins. The NMI is interrupt 2. Other external interrupts share the INTR pin via one or more 8259A Programmable Interrupt Controllers, which can map each interrupt to a unique interrupt ID in the range 32–255.
- Processor. When the processor detects a condition that it cannot handle, it communicates this fact by causing an interrupt with an ID in the range 0–16 (except for interrupt 2, which is the NMI).
- Software. Programs can generate signal events by executing the instructions INT *n* and INTO. With INT *n*, the value of *n* can be any interrupt indentifier in the range 0–255. This gives software the ability to simulate hardware interrupts as well as the ability to cause interrupts that are not directly associated with hardware events. (Note that many software systems use the software interrupt to call on operating-system services. With the iAPX 286, an interlevel CALL through a CALL gate serves this purpose.)

When an interrupt occurs, the processor uses the interrupt identifier as an index into the interrupt descriptor table (IDT).

Enabling and Disabling Interrupts

The interrupt flag (IF) controls whether the processor immediately reacts to external events. When reset, IF masks out signals presented to the INTR pin. It has no effect on NMI, on processor-detected exceptions, or on software signals (INT and INTO).

To set IF, use the STI instruction (ENABLE statement in PL/M-286); to reset IF, use CLI (DISABLE).

Interrupt Descriptor Table

The IDT associates each interrupt identifier with a descriptor for the instructions that process the associated event. The IDT is similar to the GDT and LDTs but is different in two important respects:

- The processor references the IDT only as the result of an interrupt.
- The only descriptors permitted in the IDT are three kinds of gate descriptors: task gates, interrupt gates, and trap gates (descriptor types 5–7, respectively).

The IDT may dwell at any location in memory. The processor locates the IDT via the IDT register. The operating system uses the instruction LIDT (load IDT) to set the IDT register. The instruction SIDT (store IDT) reads the contents of the IDT register. There can be only one IDT, but the operating system can use the LIDT instruction to substitute another array of gate descriptors.

Interrupt Tasks and Interrupt Procedures

In response to an event, the processor interrupts the currently executing task and begins executing the instructions identified by the IDT gate descriptor that is associated with the event. The instructions that execute as the result of the event may either be

- A task other than the current task
- A procedure within the current task

If the descriptor indexed by the interrupt identifier is a task gate, which points to a task state segment, then the processor causes a task switch. Figure 6-1 illustrates the links that identify the interrupt task. Chapter 4 discusses the mechanisms associated with task switching and considers the impact that hardware task switching has on the operating system's task scheduler.

If the descriptor indexed by the interrupt identifier is either an interrupt gate or a trap gate (which point to executable segments), then no task switch occurs. Instead the processor behaves similarly to the way it would if the current task had called the indicated procedure via a call gate. Figure 6-2 illustrates the links that identify the interrupt procedure. The iAPX 286 protection mechanism requires either that the target segment have a privilege level numerically less than or equal to CPL or that the target segment be conforming. If one of these conditions is true, then the indicated procedure begins executing in the current task. The major mechanical difference between invoking a procedure by an interrupt and invoking by a CALL is that, with an interrupt, the processor pushes the flag word onto the stack of the invoked procedure before the return address (as illustrated in figure 6-3) and clears the single-step flag (TF).

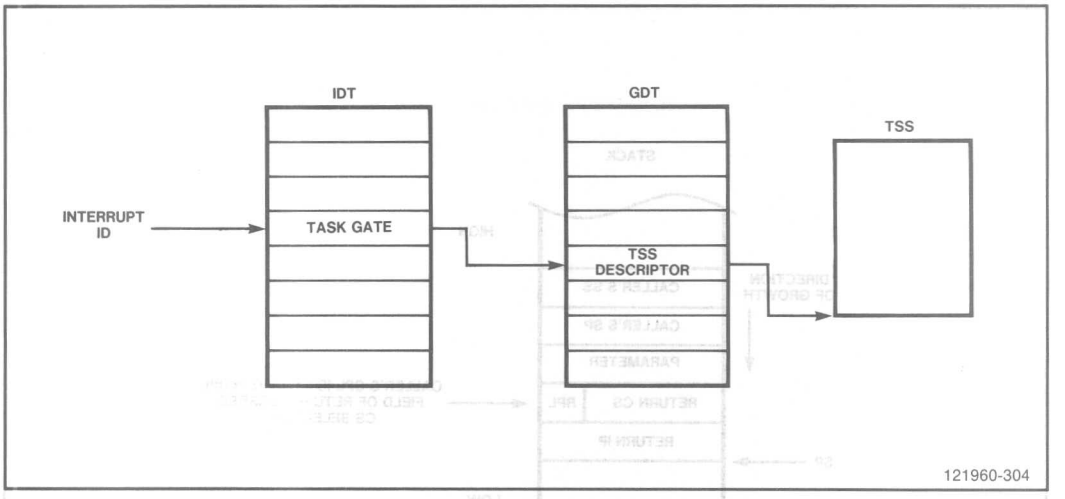


Figure 6-1. Interrupt Vectoring for Tasks

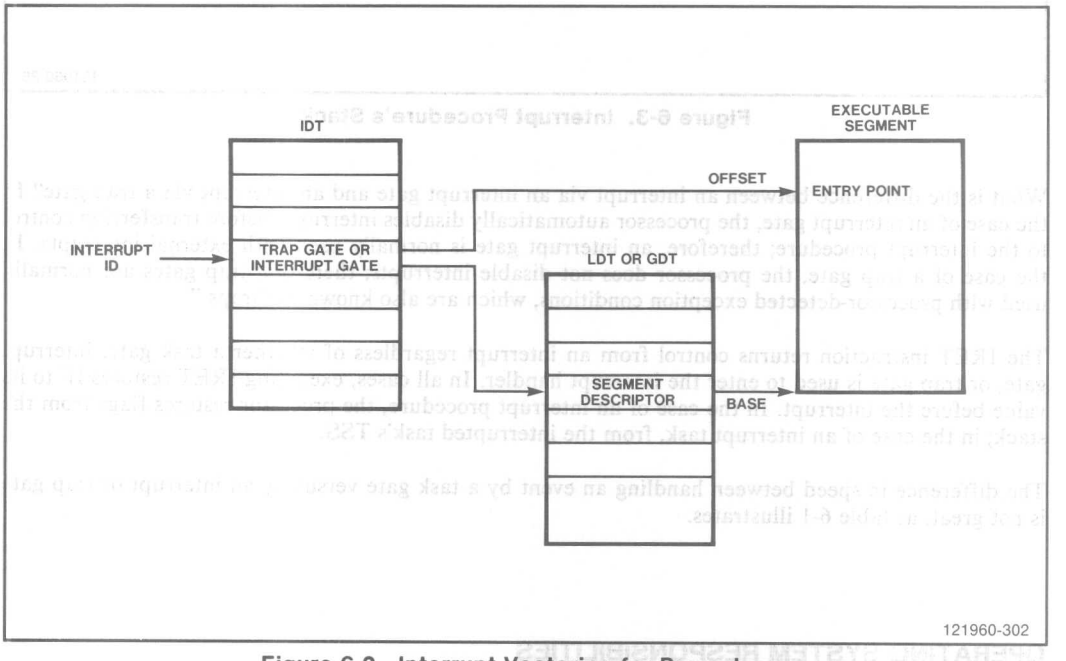


Figure 6-2. Interrupt Vectoring for Procedures

Note that if an interrupt occurs while a privilege-level 0 (PL-0) procedure is executing, an attempt to transfer to a less privileged level violates protection rules. (The same protection rules apply as for a CALL to a less privileged segment.) In general it is impossible to predict when an interrupt occurs; therefore, it is equally impossible to avoid a protection violation when a less privileged procedure has an interrupt gate or trap gate in the IDT.

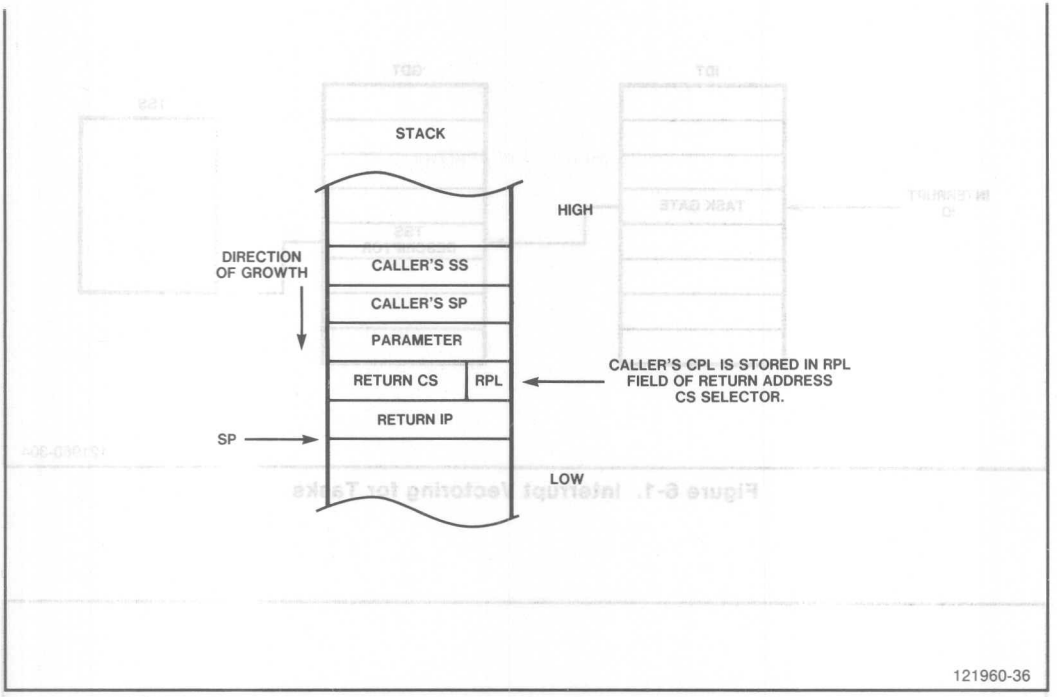


Figure 6-3. Interrupt Procedure's Stack

What is the difference between an interrupt via an interrupt gate and an interrupt via a trap gate? In the case of an interrupt gate, the processor automatically disables interrupts before transferring control to the interrupt procedure; therefore, an interrupt gate is normally used with external interrupts. In the case of a trap gate, the processor does not disable interrupts; therefore, trap gates are normally used with processor-detected exception conditions, which are also known as "traps."

The IRET instruction returns control from an interrupt regardless of whether a task gate, interrupt gate, or trap gate is used to enter the interrupt handler. In all cases, executing IRET restores IF to its value before the interrupt. In the case of an interrupt procedure, the processor restores flags from the stack; in the case of an interrupt task, from the interrupted task's TSS.

The difference in speed between handling an event by a task gate versus by an interrupt or trap gate is not great, as table 6-1 illustrates.

OPERATING SYSTEM RESPONSIBILITIES

Given the number of hardware features that automate event handling, you might wonder what is left for the operating system to do. In fact, for static systems in which interrupt tasks do not call on operating system functions and in which there is no need to change the IDT, the operating system need not concern itself with interrupts. Few applications are so simple, however. The following sections discuss some extensions to the processor's event-handling features that you may find useful in your operating

Table 6-1. Interrupt Response Time

Operation	Response Time (in number of clock cycles)	
	Task Gate	Interrupt or Trap Gate (to different PL)
Hardware interrupt processing	167	78
Save registers PUSHA PUSH DS PUSH ES	done by task switch	 17 3 3

Initialize registers
MOV AX,SEG item-1
MOV DS,AX
MOV AX,SEG item-2
MOV ES,AX
	done by task switch	
		2
		17
		2
		17

Total Clocks	**167**	**139**

system. The common goal in all these extensions is to structure the software so that, when an event occurs, the hardware can handle interrupt administration chores automatically within the efficiency and protection requirements of the application.

Manage IDT

Many applications require the ability to change the association between an event and the procedure or task associated with it. Even relatively static systems require this ability if interrupt procedures and interrupt tasks are not loaded until after system initialization. Only PL-0 procedures can effect run-time changes to the IDT because the data-segment alias for the IDT should have PL 0. The techniques for changing the IDT are similar to those already illustrated in Chapter 2 for the GDT and LDTs.

Switch Scheduling Modes

An application may include tasks that run sometimes as interrupt tasks and other times as normal tasks under the supervision of the operating system's scheduler. Examples of such situations include the following:

- An interrupt task calls on operating system services that might force the task to wait (for example, RECEIVE_MESSAGE).
- The task loader (in a dynamic system) does not distinguish between interrupt tasks and regular tasks, leaving it up to the task itself to request that the operating system attach it to an interrupt.

If such situations can arise, the operating system must

- Keep track of whether a task is in hardware-scheduled mode or in software-scheduled mode
- Provide services to switch a task between hardware-scheduled and software-scheduled mode

Note, however, that an interrupt task that is in software-scheduled mode cannot service interrupts. If it is possible for further interrupts of the same type to occur during the time the interrupt task is software-scheduled, then switching to software-scheduled mode is not such a good idea. An alternative strategy is to allocate interrupt-servicing duties among two or more tasks: one hardware-scheduled and the others software-scheduled. The hardware-scheduled task responds to the interrupt and invokes one of the software-scheduled tasks through a mechanism such as message-passing as discussed in Chapter 5. If there are any delays in servicing that interrupt, it is one of the software-scheduled tasks that waits, not the hardware-scheduled task.

Manage Interrupt Controller

Intel's 8259A Programmable Interrupt Controller (PIC) is key system resource in a multitasking system. The 8259A PIC is a flexible device that gives the main processor the ability to service up to 64 external events via the processor's single INTR pin. The 8259A PIC gives software control over such critical parameters as the relative priorities among interrupts and the means for acknowledging interrupts. Correct operation of the system requires proper use of the interrupt controller. For the operating system to manage this critical resource is only consistent with the protection features of the iAPX 286.

At system initialization the operating system may initialize the 8259A PIC according to system configuration and the needs of the application. Initialization may include

- Setting the 8259A to operate in iAPX86 mode
- Setting up master/slave relationships when the hardware configuration includes multiple PICs
- Specifying whether interrupts are triggered by edge or by level
- Setting interrupt priority mode: rotating or masked
- Determining whether priority is fully nested (if priority is set by masking)
- Determining whether interrupts are acknowledged automatically or by explicit EOI command

Refer to the *Component Data Catalog* for details of these and other features of the 8259A PIC.

If you choose an interrupt policy in which the 8259A automatically determines interrupt priority and automatically acknowledges interrupts, then there may be no need, after initialization, for either the operating system or interrupt tasks and procedures to deal with the 8259A. If, on the other hand, you choose a dynamically changing priority scheme (whether by specific rotation or by mask commands) or explicit end-of-interrupt commands, then you must also choose whether run-time control of the 8259A is the responsibility of the interrupt tasks and procedures or of the operating system.

For the operating system to maintain run-time control over the 8259A PIC, it may provide a procedure such as the following that applications programs CALL instead of executing the IRET instruction.

```
WAIT_FOR_INTERRUPT   PROC FAR

                IRET ; Switch to task on back-link
;               Execution resumes here upon interrupt
```

```
;                   Send interrupt mask to PIC
                    RET   ; Return to application procedure

WAIT_FOR_INTERRUPT  ENDP
```

The operating system executes the IRET instruction within WAIT_FOR_INTERRUPT. WAIT_FOR_INTERRUPT would need to run as a privileged procedure within the calling task. With this approach, the operating system has control just before the task begins to wait for an interrupt as well as when the task begins to execute after the interrupt occurs. The operating system then has the opportunity to send interrupt masks, end-of-interrupt commands, and specific priority rotation commands, as appropriate for the application.

Provide Task-Level Interrupt Procedures

GDT interrupt procedures (i.e., those invoked via interrupt gates and trap gates in the IDT that point to executable-segment descriptors in the GDT) provide a *global* mechanism for a task to react to events. The mechanism is global in the sense that one set of interrupt procedures applies to all of the tasks in the system. For example, suppose a GDT interrupt procedure handles the "divide error" exception. Then, a divide error in task A is handled by the same procedure as a divide error in task B because there is just one gate for divide errors. There is often a need, however, for one task to take different action than that taken by other tasks. For example, task A may need to terminate in case of a divide error, while task B may need to continue.

INTERRUPT DISTRIBUTION

The software system designer has a choice of mechanisms that make it possible for different tasks to have different interrupt procedures for a given interrupt type

1. LDT-based interrupt procedures
2. An interrupt distributor

You must deploy alternative 1 carefully. If a trap or interrupt gate in the IDT contains a selector for an LDT slot, then there must be a system-wide convention that every LDT will have an appropriate descriptor in that slot. When the interrupt occurs, the processor uses the LDT of the current task to locate the interrupt procedure.

Alternative 2 demands less from convention, but more from the operating system and the processor. With this approach, each task has (in the task database) a table that is its own private analog of the IDT. The operating system supplies a GDT interrupt procedure that merely indexes the current task's handler table to find a pointer to the appropriate handler procedure. If the task does not supply an interrupt procedure for a specific interrupt, the operating system can invoke a default procedure.

CONFORMING INTERRUPT PROCEDURES

For certain interrupt procedures (for example, a divide-error exception handler that substitutes a fixed value for the quotient), the appropriate privilege level at which to run the interrupt procedure is the same as that of the interrupted procedure. In these cases, the interrupt procedure can be placed in a conforming segment. For interrupt procedures in conforming segments, the processor automatically sets CPL to the DPL of the segment containing the interrupted procedure. Note, however, that this

technique does not apply to procedures called by an interrupt distribution procedure (because the distribution procedure always runs at PL 0).

"OUTWARD CALL"

An application may contain some interrupt procedures that should run at a fixed privilege level that is greater than zero.

The processor, on the other hand, prevents calls from PL *n* to PL *m* if $n<m$. For privilege-checking purposes the processor treats interrupts to procedures as calls. It is a privilege violation if the interrupt procedure resides in a segment that has a DPL numerically greater than that of the interrupted procedure. Since interrupts may occur at arbitrary times, it is possible for CPL to be less than the DPL of the interrupt procedure, which would be an exception.

A similar problem results when an interrupt distribution procedure (which runs at PL 0) attempts to call a less privileged procedure identified in the task's interrupt handler table. The problem results even if the interrupt distributor attempts to simulate a conforming interrupt procedure by using the interrupted procedure's CPL as the RPL value in the selector of the interrupt handler.

The operating system can employ the *shadow task* strategy to overcome this contradiction. Figure 6-4 outlines the shadow task strategy for invoking a lesser-privileged procedure from PL 0. Only a PL-0

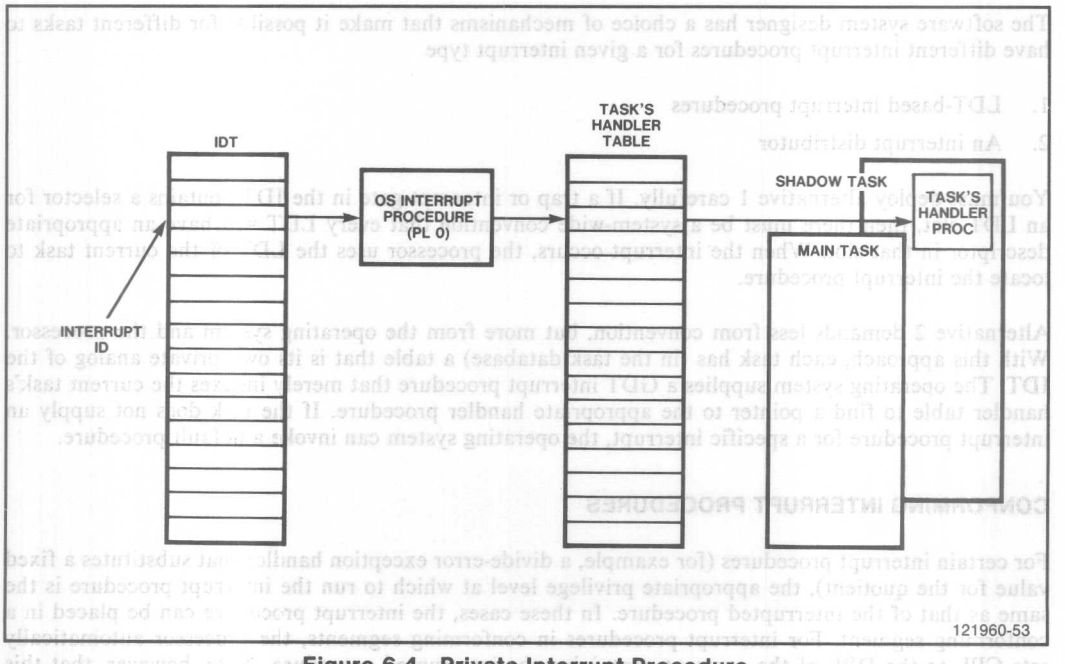


Figure 6-4. Private Interrupt Procedure

procedure such as the interrupt distribution procedure described previously can implement this mechanism. Instead of calling the handler procedure (which could be a privilege violation), the operating system's interrupt procedure

- Creates a shadow task containing the handler procedure
- Sets the CS and IP fields of the shadow task's TSS to the entry point of the handler procedure
- Calls the shadow task

An IRET instruction in the interrupt handler procedure returns control to the interrupt distributor procedure in the main task.

Creating a shadow task involves simply allocating space for the new TSS and initializing it. Setting the LDT field of the new TSS to the same value as in the main task's TSS lets the shadow task have access to all the same segments as the main task, including the segment containing the handler procedure. The operating system may create the shadow task either at the time the main task is created or dynamically, at the time the interrupt occurs.

Provide Software Signals

In some applications there is a need for one task to send a signal to another task that is not waiting for the signal. The "quit" signal in an interactive system is an example. For the target task to respond quickly to the signal, the signal must trigger some form of interrupt mechanism. The mechanisms described previously for private interrupt procedures have the appropriate features: each task can define the action to take upon receipt of the signal, and the signal handler can run at a restricted privilege level. Additional components needed to implement software signals include

- Extended task handler table with entries for each possible software signal
- Operating system procedure that any task can call to send a signal to another task

A software signalling mechanism is also a convenient way for the operating system to signal tasks. This method is particularly suited to

- Reporting exception conditions detected by the operating system
- Giving a task the chance to put its affairs in order before the operating system terminates it.

Handling Exception Conditions 7

7 Handling Exception Conditions

CHAPTER 7
HANDLING EXCEPTION CONDITIONS

When the processor recognizes a condition that it cannot handle, an *exception condition*, operating system or applications software must temporarily take control and dispose of the condition. Exception conditions are also known as "faults."

FAULT MECHANISM

The processor reports an exception condition by causing one of a predefined set of interrupts; one interrupt vector is associated with each exception condition that the processor recognizes. As with any interrupt, either a procedure or a task can field a fault. Faults resemble other interrupts, but they differ in two significant ways:

- You cannot disable a fault.
- For certain faults, the processor pushes an error code onto the stack of the fault handler (whether procedure or task) to help with recovery.

FAULT RECOVERY

When a fault occurs, the fault handler has three possible ways of dealing with the exception:

- Ignore it and continue execution of the task.
- Fix the problem and retry the faulting instruction.
- Kill the faulting task.

Ignoring an exception is not generally advisable. Killing a task is sometimes unavoidable, but, for critical tasks, the handler should make every effort to recover from the exception. In many cases, the iAPX 286 helps the exception handler identify the faulting instruction and the conditions that caused the fault.

Locating the Faulting Instruction

Usually, a fault handler can locate the faulting instruction either via the return pointer on the stack (if the exception handler is an interrupt procedure) or via the IP stored in the TSS of the faulting task (if the exception handler is an interrupt task). This stored value of the IP is used to return control to the interrupted task, but the exception handler can also use the stored IP value to examine the faulting instruction. There are three cases to consider:

- The stored IP value points to the location of the faulting instruction (including all prefixes). This is the normal case.
- The stored IP value points to the location of the *next* instruction.
- The stored IP value is unrelated to the fault. This occurs, for example, with 80287 instructions (which execute in parallel with 80286 instructions), or when the 80286 processor discovers a fault while attempting to handle an external interrupt.

Error Code

With exceptions that may relate to a specific segment, the processor pushes an error code onto the stack of the exception handler (whether procedure or task). Figure 7-1 illustrates the error code. The format of the error code resembles that of a selector. However, instead of an RPL field, the error code contains two one-bit items:

- The processor sets the EX (external) bit when the fault does not directly result from an action of the task. Occurrence of this condition generally indicates a "system" problem as opposed to an "application software" problem.
- The processor sets the I (IDT) bit if the index portion of the error code refers to a gate descriptor in the IDT. When the I bit is set, the handler can ignore the TI bit. If the I bit is reset, then the TI bit identifies either the GDT or an LDT, just as in a selector.

The index field identifies the descriptor associated with the exception (if any).

In some cases the error code on the stack is null, in which case all bits in the word are zero. For some faults, the handler can gain additional information about the fault by determining whether the error code is null.

APPLICATION INTERFACE

Since some of the actions appropriate for exception conditions depend on the requirements of individual application programs, the operating system may need to provide an application interface to the exception handling system. Chapter 6 discusses mechanisms for doing so.

EXCEPTION CONDITIONS

The action appropriate to each type of exception depends both on the type of exception and the needs of the application. This section provides details for each type of exception. Some of the exception conditions are identified by a two-character mnemonic that some other Intel literature uses.

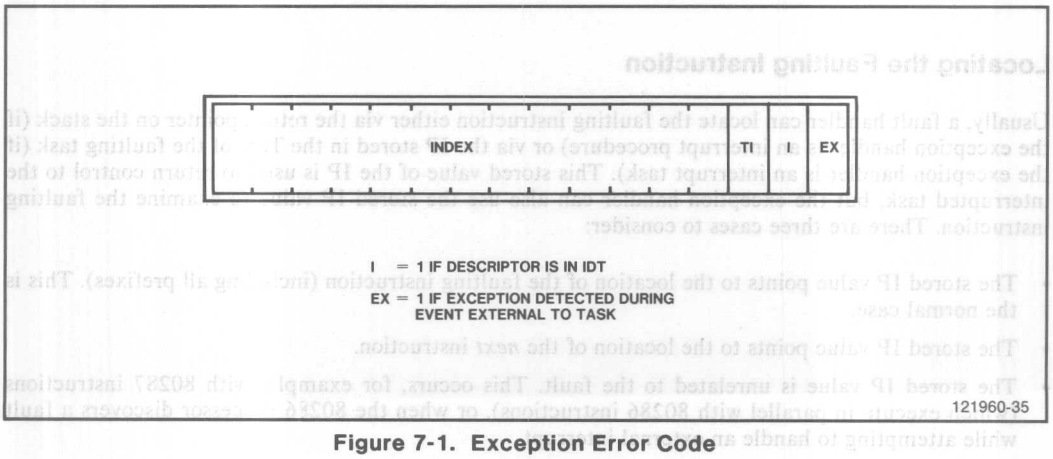


Figure 7-1. Exception Error Code

Interrupt 0—Divide Error

This exception may occur during DIV (unsigned divide) and IDIV (integer divide) instructions. The processor causes a divide-error exception in case of division by zero or quotient overflow.

A divide-error handler might, for example, replace the quotient with a predetermined value and permit the faulting task to continue. The return pointer indicates the first byte of the divide instruction, so the handler must increment the return pointer before returning.

Interrupt 1—Single Step

The processor causes a single step exception at the end of every instruction when the TF (trap flag) is set. When the processor invokes any interrupt procedure, it saves the flags and resets the TF so that the exception handler does not cause a single-step exception. Executing IRET at the end of the exception handler restores TF.

The exception handler for this exception typically belongs to a debugging system. Single stepping is a valuable debugging tool. It allows the exception handler to act as a "window" into the system through which you can observe task operation instruction by instruction. A single-step handler may, for example, display register contents, the value of the instruction pointer, key variables, etc., as they change after each instruction.

The single-step exception handler should take care, however, not to violate the system's protection goals by its actions. To protect the operating system from the debugging activities of an applications programmer, the debugger should not give access to the more privileged levels. The debugger can check, either before or after each instruction, whether the instruction causes a control transfer to a prohibited level. After such a transfer, the return pointer identifies the next instruction in an accessible segment. The debugger can set a breakpoint at that instruction and suspend single stepping until the breakpoint trap occurs.

Interrupt 3—Breakpoint

The INT 3 instruction causes this exception. The INT 3 instruction is one byte long, which makes it easy to insert a breakpoint anywhere in an executable segment. The operating system or a debugging subsystem can use a data-segment alias for an executable segment to place an INT 3 instruction anyplace where it is convenient to arrest normal execution so that some sort of special processing can be performed. Debuggers typically use breakpoints as a way of displaying registers, variables, etc., at crucial points in a task.

The saved IP value points to the next instruction. If a debugger has replaced a planted breakpoint with a valid opcode, it must subtract one from the saved IP value before returning.

Interrupt 4—Overflow

This exception occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set. Since signed arithmetic and unsigned arithmetic both use the same arithmetic instructions, the processor can not tell which is intended and therefore does not cause an overflow exception. Instead it merely sets OF when the results, if interpreted as signed numbers, would be out of range. When doing arithmetic on signed operands, careful programmers and compilers either test OF directly or use the INTO instruction.

An exception handler usually terminates the faulting task; however, in some applications it is feasible to place a "maximum value" in the receiving operand and permit the task to continue. The return pointer indicates the next instruction.

Interrupt 5—Bound Check

This exception occurs when the processor, while executing a BOUND instruction, finds that the operand exceeds the specified limits. This condition usually indicates a programming error. The safest action for the handler is to terminate the faulting task. In some applications, however, it is feasible to "adjust" the erroneous operand. The return pointer points to the beginning of the faulting instruction.

Interrupt 6—Undefined Opcode (UD)

This exception occurs when the processor detects an invalid operation code in the instruction stream. Such an exception may occur, for example, if a programmer mistakenly causes a jump to read-only data in an executable segment. This exception has several variations:

- The first byte of the instruction is completely invalid; for example, 64H.
- The first byte of the instruction indicates a two-byte opcode, but the second byte is invalid; for example, 0FH followed by 0FFH.
- One of the operands of the instruction is not valid for use with the opcode.
- The opcode extension in the second byte of an instruction contains a value that is invalid for use with the opcode; for example, opcode 0F6H with *xx*001*xxx*B in the second byte.
- The opcode requires a memory operand, but the operand actually indicates a register, for example, LGDT AX.

The offending opcode is invalid, so the handler should not restart the instruction.

You can use this exception to implement extensions of the iAPX 286 instruction set. The exception handler would interpret the instruction and advance the return pointer beyond the extended instruction before returning.

Interrupt 7—Processor Extension Not Available (NM)

This exception occurs in either of two conditions:

- The processor encounters an ESC (escape) instruction, and the EM (emulate) bit of the machine status word (MSW) is set.
- The processor encounters a WAIT instruction, and both the MP (math present) and TS (task switched) bits of the MSW are set.

Refer to Chapter 12 for details concerning the 80287 Numerics Processor Extension.

Interrupt 8—Double Fault (DF)

This exception occurs when the processor detects an exception while trying to invoke the handler for a prior exception, for example:

- The code segment containing the exception handler is marked "not present."
- Invoking the exception handler causes a stack to overflow.

The processor pushes a null error code onto the stack. If this or any other action while invoking the double-fault handler causes an additional fault, the processor shuts down. To avoid the catastophe of a shut-down, the double-fault handler must be a separate task (so that pushing the error code does not cause a stack fault) and must always be present (so that invoking it does not cause a "not present" fault).

Recovery is sometimes possible by eliminating the cause of the second exception and re-executing the faulting instruction so that the original fault can be handled appropriately. The greatest difficulty lies in identifying the cause of the second exception. Often, however, a double-fault condition indicates a serious error, and the faulting task should be terminated.

Interrupt 9—Processor Extension Segment Overrun

This exception occurs when a memory operand of an 80287 instruction has a segment-limit violation. Since the 80287 executes in parallel with the 80286, the occurrence of this exception may not relate directly to the instruction stream being executed by the current task. A task switch may have occurred since the 80287 began executing the instruction. Even if the interrupted task is the correct task, its IP may have been advanced by several instructions beyond the 80287 instruction. Refer to Chapter 12 for more information about this exception.

Interrupt 10—Invalid TSS (TS)

This exception occurs when the processor detects any of the following abnormalities in a TSS:

1. The value of the limit field of the descriptor to the TSS is too small (discovered during a task switch).
2. The LDT indicated in the TSS is invalid or not present (task switch). Note that a null LDT selector does not cause an exception during a task switch.
3. One of the segment register fields (SS, CS, DS, or ES) in the TSS is invalid (task switch).
4. One of the privileged-stack selectors is invalid (interlevel CALL).
5. The back-link selector is invalid (intertask IRET).

This exception does *not* occur when a segment-register field or the back link is marked "not present" but is otherwise valid. A "not present" exception occurs in this case. If, during an interlevel CALL, a privileged-stack selector in the TSS points to a descriptor marked "not present," then a stack exception occurs.

The handler for this exception must be a separate task, invoked via a task gate in the IDT. The processor pushes an error code onto the stack of the handler. The error code either identifies the faulty TSS (case 1) or contains the faulty selector from the TSS (cases 2–5). The instruction causing the fault can be restarted. An IRET at the end of the exception handler causes the faulting instruction to execute again.

A few of the conditions that can cause this exception are recoverable. If the system incorporates virtual memory, the solution for a not present LDT may be to bring it in from virtual store. In the case of an invalid back-link, you can make the assumption that the NT flag was set by mistake, give control to the scheduler, and hope....

Interrupt 11—Segment Not Present (NP)

This exception occurs when the processor detects that the present bit of a descriptor is reset. It may occur in any of these cases:

- While loading the CS, DS, or ES registers, but not while loading the SS register (a stack fault occurs in that case).
- While loading the LDT register with an LLDT instruction, but not while loading the LDT register during a task switch operation (the "invalid TSS" fault occurs in that case).
- While attempting to use a gate that is marked "not present."

An operating system typically uses the "not present" exception to implement a virtual memory system. Refer to Chapter 9 for more information on virtual memory.

The processor pushes an error code onto the stack of the exception handler. The error code contains the selector of the descriptor that is marked "not present."

A "not present" indication in a gate descriptor usually has special significance for the operating system. For gates in the IDT, the present bit may serve as a sign that the interrupt task is in software scheduled mode and temporarily unable to service an interrupt. If an interrupt arrives in this case, there may be an error either in the device that generates the interrupt or in the handling of the Interrupt Mask Register of the 8259A PIC. (Refer to Chapter 6 for more information on interrupt handling). For gates in the GDT or LDTs, the present bit may serve to signal an unresolved linkage. (Refer to Chapter 11 for information on binding.)

The instruction that causes a "not present" fault is restartable (except in the case of a task switch). Execution of an IRET by the exception handler causes the processor to execute the faulting instruction again. When the processor detects the "not present" exception while loading CS, DS, or ES during a task switch, the exception occurs in the new task, and the return pointer points to the first instruction of the new task.

Interrupt 12—Stack Exception (SS)

This exception occurs in either of two general conditions:

- As a result of a limit violation in any operation that refers to the SS register. This includes stack-oriented instructions such as POP, PUSH, ENTER, and LEAVE as well as other memory references that implicitly use SS (for example, MOV AX,[BP+6]). ENTER causes this exception when the stack is too small for the indicated local variable space. An interlevel CALL references two stacks; a stack-limit exception can result from either of them.
- When attempting to load the SS register with a descriptor that is marked "not present" but is otherwise valid. This can occur in a task switch, an interlevel CALL, an interlevel return, or a MOV or POP instruction to SS.

The processor pushes an error code on the stack of the exception handler. If the exception is due to a not-present stack segment or to overflow of the new stack during an interlevel CALL, the error code contains a selector to the segment in question (the exception handler can test the present bit in the descriptor to determine which exception has occurred); otherwise the error code is null.

The instruction that causes a stack fault is usually restartable. Execution of an IRET by the exception handler causes the processor to execute the faulting instruction again. The one case that is not restartable is a PUSHA or POPA intruction that attempts to wrap around the 64K boundaries of a stack segment. This condition is identified by one of the values FFFEH, FFFFH, 0000H, or 0001H in the saved SP.

When the processor detects a stack fault while loading SS during a task switch, the exception occurs in the new task, and the return pointer has the value that the IP field of the new TSS held at the time the task switch began.

When stack overflow causes the exception, the stack fault handler can increase the size of the stack segment (up to a maximum of 64K) and permit the faulting task to continue. When the exception is due to a stack segment not being present, recovery action resembles that of the "not present" fault handler.

Interrupt 13—General Protection Exception (GP)

All protection violations that do not cause another exception cause a general protection exception. This includes (but is not limited to)

- Exceeding segment limit when using DS, ES, or CS
- Exceeding segment limit when referencing a descriptor table
- Jumping to a data segment
- Writing into a read-only segment or an executable segment
- Reading from an execute-only segment
- Loading the SS register with a read-only descriptor (unless the selector comes from the TSS during a task switch, in which case a TSS exception occurs)
- Loading DS, ES, or SS with the descriptor of a system segment
- Loading DS, ES, or SS with the descriptor of an executable segment
- Loading CS (by means of a CALL, JMP, or interrupt) with the descriptor of a data segment
- Accessing memory via DS or ES when it contains a null selector
- Switching to a busy task
- Violating privilege rules

The processor pushes an error code onto the exception handler's stack. If loading a descriptor caused the exception, the error code contains a selector to the descriptor; otherwise the error code is null.

The return pointer points to the beginning of the faulting instruction. Recovery may be possible, but because programming errors cause most of the conditions leading to a general protection exception, recovery may not be worth the trouble required to identify the cause.

A string instruction (any variant of INS, OUTS, MOVS, STOS, SCAS, or CMPS) with a repeat prefix (REP, REPE, or REPNE) is not restartable if it causes a segment limit violation. Check whether SI or DI is near the segment limit.

Interrupt 16—Processor Extension Error (MF)

The 80286 causes this exception when it detects a signal from the 80287 on the 80286's $\overline{\text{ERROR}}$ input pin. The 80286 tests this pin only at the beginning of certain floating-point instructions and when it encounters a WAIT instruction while the EM bit of the MSW is reset (no emulation).

Refer to Chapter 12 for more information on this exception.

Interrupt 17—Run-Time Exceptions

Intel's run-time support software uses this interrupt to communicate exception conditions regarding range checks and procedure stack overflow. Applications should avoid using this interrupt for any other purpose.

RESTARTABILITY SUMMARY

Table 7-1 summarizes the information pertinent to restarting the faulting instruction.

Table 7-1. Restart Conditions

Vector	Exception	Return Address Relative to Faulting Instruction	Restartable?	Error Code?
0	Divide error	First byte	Yes	No
1	Single step	Next instr.	No	No
3	Breakpoint	Next instr.	—	No
4	Overflow	Next instr.	No	No
5	Bound check	First byte	Yes	No
6	Undefined opcode	First byte	No	No
7	Processor extension not available	Unrelated	Yes	No
8	Double fault	First byte	No	Yes (always null)
9	Processor extension segment overrun	Unrelated	No	No
10	Invalid TSS	First byte	Yes	Yes
11	Segment not present	First byte	Yes	Yes
12	Stack exception	First byte	Yes	Yes
13	General protection	First byte	No	Yes
16	Processor extension error	Unrelated	—	No

Input / Output 8

8 Input\Output

CHAPTER 8
INPUT/OUTPUT

Many of the concepts previously introduced in this book apply to the design of the I/O subsystem of an operating system; in addition, the iAPX 286 has several I/O features of special interest.

The functions typically performed by an I/O subsystem include

- Centrally implementing and standardizing I/O logic so that all application programs can share it.
- Providing a uniform, high-level interface by which application procedures can request I/O services (as a minimum, the functions READ and WRITE). A more sophisticated system may need a full, file-oriented set of interfaces such as Intel's Universal Development Interface (UDI).
- Administering device naming. Often this includes transforming logical device identifiers into physical identifiers, so that applications that use the logical identifiers can maintain independence from physical devices.
- Managing use of memory resources for I/O buffers.
- Managing sharing of physical devices. Often this reduces to giving one task exclusive use of a device (for example, a printer) until the task relinquishes it. With disk devices, tasks can usually share the device as long as each uses a different set of disk addresses. A sophisticated database-management system may require unlimited disk sharing (the database system assumes responsibility for block-level synchronization, deadlock detection, and recovery).
- Providing device-driver procedures to deal with the vagaries of various I/O devices.
- Optimizing I/O efficiency. This might include any of these techniques: blocking, buffer pooling, automatic seek-ahead, reduction of disk arm movement.

This discussion of I/O classifies physical I/O operations thus:

- Direct I/O, in which the 80286 processor itself communicates directly with the peripheral device. Direct I/O breaks down further into
  a. Memory-mapped, in which I/O is triggered by processor instructions that reference certain memory locations.
  b. I/O-mapped, in which special I/O instructions cause the processor to do I/O.
- Indirect I/O, in which an external processor (such as the Intel 8089 I/O Processor) performs the I/O.

I/O AND PROTECTION

The concept of protection as applied to I/O deals not only with the memory used in I/O operations but also with the right to execute I/O operations.

I/O Privilege Level (IOPL)

You can limit to specific privilege levels the right to execute I/O and I/O-related instructions. IOPL (a two-bit field in the flag word) restricts a task's right to execute any of these instructions:

IN	input
INS	input string

OUT	output
OUTS	output string
STI	set interrupt flag (enable interrupts)
CLI	clear interrupt flag (disable interrupts)
LOCK	lock bus

When interpreting any of these restricted instructions, the processor compares CPL to IOPL. If CPL exceeds IOPL, the processor causes a general protection exception and does not carry out the instruction.

Only a privilege-level 0 (PL-0) procedure (i.e., the operating system) can change IOPL. There is no instruction that explicitly affects IOPL; however, any of the operations that load the flag word can, in some cases, change IOPL. The only mechanisms for changing the flag word are

- A task switch
- The POPF (pop flags) instruction
- IRET

When CPL is greater than zero, the POPF instruction does not change IOPL, even though it changes other flags in the flag word. The processor issues no error indication when this occurs. A task switch loads the flags from the Task State Segment (TSS). As long as the operating system does not make data-segment aliases for the TSS available to less privileged levels, only the operating system can change IOPL in the TSS.

For maximum protection, the procedures of an I/O subsystem that run in the calling task should run at a protection level numerically greater than the operating-system kernel but less than applications procedures. IOPL can then include the I/O subsystem but exclude applications procedures. Used this way, IOPL forces less privileged application procedures to call on I/O subsystem procedures for I/O functions, thereby giving the operating system control over many I/O operations.

Tasks that deal primarily with I/O (device drivers, for example) may have an IOPL value as great as three. If that is the case, all procedures in the task have access to I/O operations, yet all four privilege levels are available to protect the procedures of the task from one another.

Controlling I/O Addresses

Protection is incomplete if not applied to memory accesses by I/O operations. IOPL does not apply to memory-mapped I/O nor to interface with intelligent controllers (because none of the restricted instructions are used). The operating system designer must make special provisions to control these I/O operations, either via the operating system or with the Builder.

HARDWARE ADDRESS CHECKING

Memory-mapped I/O is subject to the segment-level protection mechanism of the iAPX 286. A task can execute a memory-mapped I/O operation only if it has access to a descriptor for a data segment that contains one of the memory addresses reserved for I/O. Giving a task descriptors for only the I/O memory addresses that it has the right to use yields a double benefit:

- The task cannot access I/O devices assigned to other tasks.
- Within the task, I/O is restricted to those procedures whose privilege level is numerically less than or equal to the DPL of the I/O memory address segments.

You can still take advantage of I/O memory address protection even though the I/O devices on the bus respond to I/O commands. A hardware address mapper can convert the processor's memory cycles for the I/O memory space into I/O cycles on the bus. The address mapper will not initiate a bus I/O cycle unless the memory operation passes the processor's protection checking.

SOFTWARE ADDRESS CHECKING AND CONVERSION

With indirect I/O, the external controller accesses I/O buffers independently of the 80286. This presents a three-fold problem to the operating system:

- Memory access by the controller is not subject to the automatic protection checking of the 80286.
- The controller (probably) uses "flat" addresses (i.e., addresses that are not relative to a base address).
- The addressing range of the controller may not coincide completely with that of the 80286.

The flow of control for indirect I/O requests must pass through operating-system procedures at PL 0 so that the operating system can

- Check memory addresses and privilege levels against information stored in the task's descriptors
- Transform base-relative addresses to a flat format recognizable by the controller

I/O AND MEMORY MANAGEMENT

An I/O subsystem can place additional requirements on the operating system's memory manager; for example:

- I/O functions may use certain dedicated memory locations (for example, the addresses used for memory-mapped I/O, and the communication blocks and buffers that external controllers expect to find a fixed locations). The memory manager must be aware of these locations and must not allocate them for other purposes.
- When buffers for external controllers are allocated dynamically, addressing limitations of the controller may require the memory manager to find space within the portion of memory that the controller can address.
- Once it allocates a segment for the use of an external controller, the memory manager must not move, delete, or swap out the segment without cooperation from the controller.

PARTITIONING I/O FUNCTIONS

To determine how best to distribute I/O functions across tasks and privilege levels, you must consider

- The opportunities for parallelism
- The needs for synchronization
- The requirements for protection

Requirements for Parallelism and Synchronization

The requirements for parallelism and synchronization in the I/O subsystem may include any of the following:

- The application procedure that requests a READ operation may run in parallel with the I/O subsystem until that procedure needs to reference the requested data, at which point it must wait until the data arrives or an exception occurs.
- The procedure that requests a WRITE operation can usually run in parallel with the I/O subsystem. Some applications require acknowlegement of the completion of a WRITE operation (in order, for example, to synchronize with a database recovery system), in which case that procedure must wait until the acknowledgement arrives.
- SEEK operations normally run in parallel with the requesting procedure.
- The I/O device can always run in parallel with some task in a multitasking system, whether it be the task that requested the I/O operation or some other task that is not waiting for I/O.
- In a simple I/O subsystem in which a device driver only manages one I/O operation at a time, the driver can simply wait until the device signals that the operation finishes. If the requesting procedure is blocked, the device driver can merely convert the I/O-complete signal into a wake-up signal for that procedure.
- In a more sophisticated I/O subsystem (for example, one in which a disk driver handles more than one spindle and more than one task can share a disk device), greatest efficiency results only when device drivers run in parallel with I/O devices as well as with requesting procedures. An I/O-complete signal from a device may arrive when the device driver is busy.

Figure 8-1 explains the symbols used in a Petri net graph. Figures 8-2 and 8-3 use Petri net graphs to illustrate two approaches to synchronization between parts of an I/O subsystem. A horizontal line represents an event of interest that occurs only under certain conditions. Circles preceding an event represent the conditions under which the event can occur. Circles after an event represent the conditions that result from occurrence of the event. A dot inside a circle is called a *token*. A token represents a condition that is in effect. An event occurs if and only if there are tokens for all its input conditions. When an event occurs, tokens flow into all its output conditions.

Figure 8-2 assumes the simple device driver mentioned previously and points out how such a driver can service only one task at a time. Figure 8-3 shows how a two-part driver can deal with more than one I/O request at a time (assuming that the I/O device can do so as well). I/O requests from applications procedures drive part A, while interrupts drive part B. Not shown by the Petri net graphs is the fact that the two parts of the driver must share information about outstanding I/O requests. Both figures show simplified device drivers to highlight the interactions among parts of the I/O subsystem; for example, a real device driver may issue multiple physical I/O commands in response to one I/O request and may retry I/O operations in case of certain error indications from the device.

Requirements for Protection

The requirements for protection in an I/O subsystem include (in addition to the protection considerations previously discussed)

- Device allocation tables and any other data used by the primary I/O interface procedures must be protected from all but those procedures privileged to do I/O, but must be available to every task in which the I/O interface procedures can run.
- Only the device driver should have access to a device's control parameters (for example, head settling time for a disk drive).

- Only the operating system and the I/O subsystem should use queues of buffers and I/O requests.
- An application procedure must not use a buffer that an I/O device is simultaneously using.

Implementation Alternatives

If descriptors for data global to the I/O subsystem (such as device allocation tables) reside in the GDT with DPL equal to the I/O privilege level, then I/O procedures can access them regardless of what

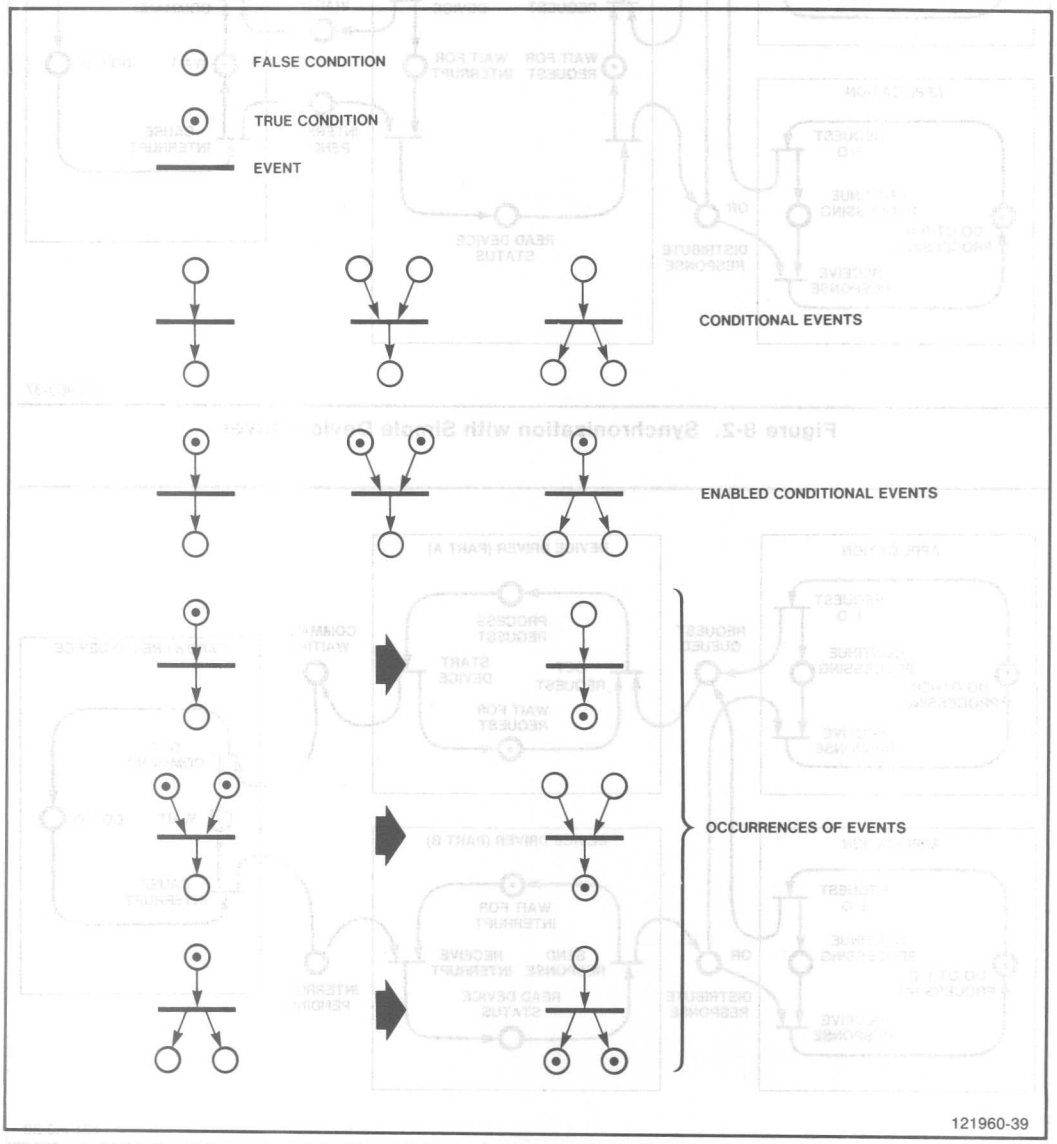


Figure 8-1. Petri Net Graph Symbols

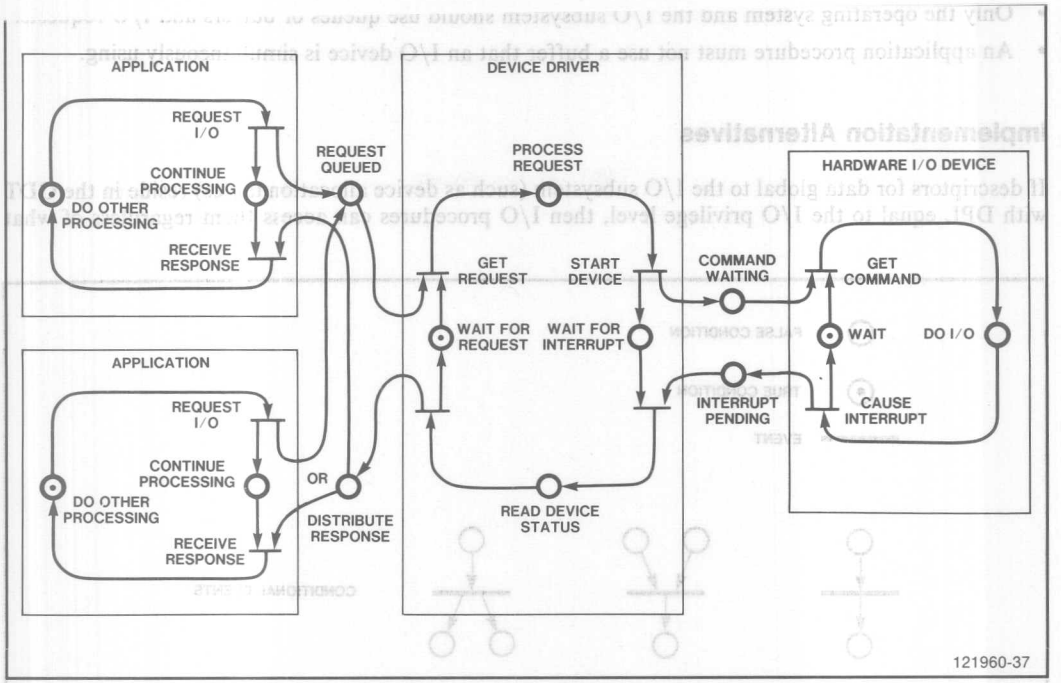


Figure 8-2. Synchronization with Simple Device Driver

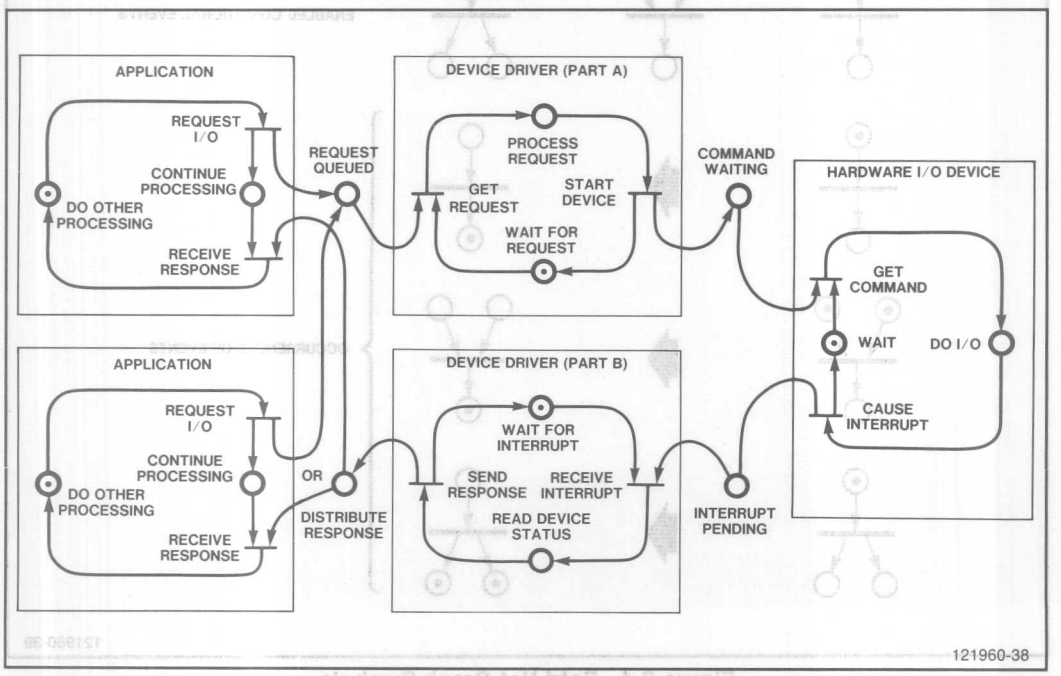


Figure 8-3. Synchronization with Two-Part Device Driver

task the I/O procedures run in. The I/O interface procedures (READ, WRITE, etc.) must have DPL equal to that of the global I/O data and therefore must have call gates at the privilege level of application procedures (PL 3). If these call gates reside in the LDTs of application tasks, then I/O interface procedures can be shared by (but limited to) those tasks that need them. The call gates can also reside in the GDT, where all tasks can access them. In either case the interface procedures run in the calling task.

The possibilities for running device drivers in parallel with the calling task suggests that they be separate tasks. Such a separation also serves to protect applications and device drivers from one another, a definite advantage because inherent complexity and frequency of change tend to place device drivers among the least reliable of operating system functions. The I/O interface procedures have the responsibility for managing the details of communication between the calling task and the device drivers.

You can implement a sophisticated device driver such as that illustrated in figure 8-3 as two cooperating tasks: one scheduled by interrupts via a task gate in the IDT, and the other scheduled by I/O requests via an operating-system mailbox facility.

It is possible to implement device drivers as interrupt procedures that run within applications tasks. Such a structure is advantageous in these cases:

- Efficiency of I/O is an overriding goal.
- Interrupts may arrive when a driver is busy.

The lack of protection inherent in such a structure becomes apparent, however, when you consider that the driver procedure that fields an I/O-complete interrupt may run as part of a task completely unrelated to the task that originally requested the I/O operation, and must run at PL 0.

Viewing the possible implementations of buffers from the perspective of the iAPX 286 architecture, the most pertinent consideration is whether a segment contains just one buffer or several buffers. An approach that uses one segment per buffer has several advantages:

- The protection mechanism of the iAPX 286 (working as it does on segment descriptors) focuses on each buffer individually.
- The selector of a buffer segment serves as a convenient buffer identifier, fitting easily into the aliasing and mailbox schemes outlined in Chapter 5.

You can avoid the potential GDT congestion that may result from having one segment (and therefore at least one segment descriptor) per buffer by storing buffer descriptors in LDTs.

When tasks share buffers (as between an application task and one or more device-driver tasks), you have a choice between

- Leaving the buffer at all times within the address spaces of all the sharing tasks
- Transferring the buffer from the address space of one task into that of another

The mailbox scheme outlined in Chapter 5 easily accomplishes the latter approach. This scheme has the advantage that the application task cannot use the buffer at the same time as I/O is in progress. The "usage privilege level" (UPL), if set to IOPL, provides additional protection by limiting mailbox access to I/O procedures. The application's requirements for I/O efficiency may, however, preclude use of mailboxes for this purpose.

Virtual Memory

9

CHAPTER 9
VIRTUAL MEMORY

The memory legitimately addressed by the tasks running on an iAPX 286 (the *virtual memory*) may exceed the actual memory available. You can use this capability to lower memory costs, substituting disk or other less expensive storage media for relatively expensive RAM. Virtual memory isolates programmers from the amount of real memory in a computer system. The system designer can trade off performance against system cost using identical software.

A system that supports virtual memory can be analyzed in terms of mechanisms and policies. The iAPX 286 has mechanisms that help your operating system manage the swapping of segments between RAM and less expensive memories. The operating system must implement additional mechanisms as well as policies for efficient use of these mechanisms in a specific application.

HARDWARE MECHANISMS

The 80286 provides the essential hardware mechanisms without which virtual memory systems would not be possible. The *segment* is the basic unit of the virtual-memory scheme, just as it is the basic unit of the real-memory scheme. In each segment descriptor, the iAPX 286 architecture provides an *accessed bit* and a *present bit* to aid the operating system in simulating the virtual memory space with available RAM.

Accessed Bit

Every descriptor for an executable segment or data segment has an *accessed flag* in the least significant bit position of the access rights byte. Each time a task loads segment register with a segment descriptor, the processor automatically sets the accessed bit in that descriptor. The processor does not automatically reset the accessed bit; software must explicitly write a zero into the accessed bit. The accessed bit has a dual function in virtual memory management:

- By testing and then resetting the accessed bit at regular intervals, the virtual-memory manager can measure how frequently the segment is being accessed.
- For writable data segments, the accessed bit (when set) indicates that the segment may have been changed.

Present Bit

Every segment descriptor has a *present flag* in the high-order bit position of the access-rights byte. The processor automatically tests this bit as it loads segment registers. If the present bit is reset, the processor causes a trap. Which trap depends on the circumstances:

- Trap 11 (Segment not Present) occurs when loading the CS, DS, or ES register with a not-present segment descriptor, when switching to a not-present TSS, when loading the Task Register by means of the LTR instruction with a not-present TSS descriptor, or when loading the LDT register with a not-present LDT descriptor.
- Trap 11 also occurs when loading CS with a gate descriptor that is marked "not present." This condition does not necessarily mean that a segment is not present, however. The operating system

may attach other meaning to the present bit of gate descriptors. Refer to the section on load-time binding in Chapter 11 for an example of an alternate use for the present bit in gate descriptors.

- Trap 12 (Stack Exception) occurs when loading the SS register with a not-present descriptor. The exception handler can distinguish this condition from other stack exceptions by examining the access rights byte of the descriptor selected by the error code.
- Trap 10 (Invalid TSS) occurs when switching to a TSS that points to a not-present LDT. The exception handler can distinguish this from other "invalid TSS" conditions by examining the access rights byte of the descriptor selected by the error code.
- Trap 8 (Double Fault) occurs if the processor, while trying to invoke an exception handler due to a previous exception, finds that the code segment containing its entry point is not present. The difficulty of distinguishing between this and other double fault conditions implies that trap 8 is to be treated as an error condition, not a normal use of the present bit.

Refer to Chapter 7 for additional information about these traps.

SOFTWARE MECHANISMS

The operating system must provide additional mechanisms for a virtual memory system: one for moving a segment from RAM to secondary storage (the swap-out manager) and one for moving a segment from secondary storage to RAM (the swap-in manager).

The swapping managers are essentially I/O modules, but there are major differences between swapping managers and the functions of a standard I/O subsystem:

- Swappers deal with executable segments, system segments, and data segments that are not normally considered I/O buffers.
- I/O performance of swappers is critical and often calls for specialized device drivers and disk space allocation stategies.

Secondary Storage Management

Virtual-memory mechanisms use a secondary storage medium, such as disk, to simulate a larger memory space than that provided in RAM. The operating system uses this secondary storage (here called the *swap space*) to store copies of those segments that are currently in the virtual space but may be eliminated from RAM.

There are two general approaches for allocating swap space for segments in dynamic systems:

- The loader can invoke a swap-space allocation procedure as it loads the segments of a task. It can at the same time write an initial image of the segment into the swap space. This is particularly useful for a segment such as an executable segment that is occasionally swapped in but may never be swapped out (when its RAM space is used for another segment) due to the fact that its contents do not change.
- The operating system may invoke the swap-space allocation procedure dynamically, either when allocating RAM for the segment or at the first time the operating system swaps the segment out.

Some operating systems may implement both approaches. A system that allocates swap space only at load-time cannot swap out certain segments; namely, segments that a bootloader creates initially and segments that the operating system creates dynamically. In many systems, this is not a problem. Often

those segments that are bootloaded are just the segments that should not be swapped out. Other operating systems may allocate many segments dynamically: stacks, mailboxes, variable-length arrays, etc. These systems may need both approaches.

In designing a dynamic swap-space allocation scheme, you should consider at what time it is best to allocate swap space. The procedure that first allocates RAM space for a segment (a loader, for example) often creates a writable data segment descriptor. Later, when the procedure has initialized the segment (for example, by writing descriptors into a segment that is to be used as an LDT), it modifies the type code in the descriptor to reflect the intended use of the segment (in this example, it would change the type to LDT). By delaying the allocation of swap space until first swap out, the operating system never allocates swap space for segments that it never swaps out (there is no need to allocate swap space for an LDT, for example, if the operating system does not support swapping of LDTs).

Once the operating system allocates swap space for a segment, it must store the swap-space address in a location that is easily accessible when it is time to swap the segment out. Two possible mechanisms for storing the swap-space address are

- In a boundary tag of the segment, if boundary tags are used. (Refer to Chapter 3 for an example of a space-management scheme that uses boundary tags.)
- In a table parallel to the descriptor table.

To determine whether to call the swap-space allocation procedure when it is time to swap a segment out, the operating system can test whether the swap-space address field contains a null value.

When a segment is not present, the operating system can use the 24-bit base-address field in the segment descriptor to store the address of the swap space in which the segment is stored. As long as the present bit of a descriptor is reset, the processor does not use the base-address field. Storing the swap-space address in the descriptor also makes the swap-space address readily accessible to the "not-present" trap handler because the error code presented to the trap handler contains a selector to the descriptor for the not-present segment.

Level Zero Support Procedures

While the swapping procedures are I/O procedures that should run at privilege levels greater than zero, protection of the system demands that highly privileged procedures carry out some details of the swapping process. Only privilege-level 0 (PL-0) procedures have the right to perform such activities as

- Create read-data or write-data alias descriptors with which the swappers can access the segments they are operating on
- Change the present bit in a descriptor
- Overwrite the base-address field of a descriptor
- Prevent the swapper from operating on segments that must remain RAM-resident
- Update all the alias descriptors for a segment with its new status

The following checklist identifies some of those segments that should remain permanently in RAM (in your application, there may be others):

- The GDT. (It is the key to all addressing operations.)
- LDTs that refer to present segments. (The processor cannot access an LDT segment without fetching its descriptor from the LDT.)

- TSSs that point to present LDTs. (You cannot switch to a task and use its LDT without referring to its TSS.)
- TSSs whose NT (nested task) flag is set.
- Certain operating-system kernel segments that are frequently referenced (for example, the segment or segments containing the scheduler's queues). (System performance may degrade excessively.)
- Segments belonging to the swapping managers.
- I/O buffers that are in use by an external device.
- Executable segments that contain the entry point of an exception handler. (An exception would result in a double fault.)

Note that, while the iAPX 286 does offer mechanisms that support swapping of TSSs and LDTs, doing so is likely to cause not-present faults in PL-0 procedures and to cause unacceptable delays in invoking interrupt tasks that are not present. For either of these reasons, the designers of an operating system may elect not to swap out TSSs and LDTs.

Swapping Managers

Swapping managers may need to distinguish between two classes of segments:

- Segments of a task superstructure (the TSS, the task database (TDB), and possibly the level-zero stack segment)
- Segments not part of a task superstructure

Swapping of the task superstructure requires that swapping managers be aware that the kernel may treat the TSS as a "subsegment" of the TDB, which may itself reside within the level-zero stack (as outlined in Chapter 4). The swapping managers should treat these segments as a unit.

Considering the complexities associated with swapping the segments of the task superstructure, it is perfectly reasonable for an operating system to simplify its virtual-memory subsystem by leaving those segments in RAM for the duration of the task.

OUT-SWAPPER

The out-swapper works best as a separate task; when the out-swapper must wait for the swapping-device I/O driver to write a segment, other tasks can continue to run, including the task whose segment is being swapped out.

The out-swapper's responsibility is to

- Mark all the descriptors for the segment "not present." The out-swapper must ensure that the present bits in all descriptors for a segment always appear consistent. It must use a semaphore or region to prevent other access to the aliases while it is changing present bits.
- Copy the swap-space address into all the descriptors for the segment (or possibly, depending on alias implementation, into a "master descriptor" that is linked to other descriptors).
- Create a temporary data-segment descriptor to give the swapper the right to read the segment. The operating system must not move or delete this segment until the out-swapper is finished with it.
- Write the segment to the swap-space allocated for it (but only if the segment is writable and its accessed bit is set).

- Return the RAM space used by the segment.
- Delete the temporary data-segment descriptor.

IN-SWAPPER

In theory, the fetch policy module, invokes the in-swapper. In practice, when the fetch policy is "on demand," the in-swapper is the "not-present" fault handler, which may also be called by the stack segment fault handler (for not present stack segments) and by the "invalid TSS" fault handler (for not present LDTs). The not-present fault handler can run as a procedure in the task that caused the fault.

There is one case, however, that such a procedure cannot handle well. A dispatcher procedure running in task A causes a not-present fault when switching to task B whose TSS is not present. If the not-present handler procedure continues running in task A, then task A must wait until task B's TSS can be swapped in. Whether this is a problem depends on the role of task A in the application. The not-present handler procedure can avoid this situation by suspending task B and sending a message to yet another task that is dedicated to swapping in TSSs.

The steps that an in-swapper procedure takes are to

- Get the swap-space address and segment size (limit) from the descriptor indicated by the error code.
- Allocate a writable data segment in RAM large enough to receive the segment.
- Copy the segment from swap space to the newly allocated RAM space.
- Update all regular descriptors for the segment with the new base address, setting the present bit and resetting the accessed bit. (The base address comes from the temporary writable data-segment descriptor.)
- Delete the temporary writable data-segment descriptor.

An in-swapper task for not present TSSs gets a message from its input mailbox that identifies the descriptor for the TSS and identifies the task to which the TSS belongs. The in-swapper task performs the same steps as an in-swapper procedure, but it must also inform the scheduler when the task is ready to run. You can avoid the additional complexities of not-present TSSs by not swapping TSSs out.

COORDINATION OF IN- AND OUT-SWAPPER

A number of interactions between the in-swapper and out-swapper present pitfalls that the operating system must avoid:

- A task may attempt to use a segment that is being swapped out. If the in-swapper does not handle this possibility, it may swap in old, erroneous data from the swap-space.
- Task A may request swapping in of a shared segment that is being swapped in for task B. If the in-swapper blithely reads in the segment again for task A, it may overwrite changes made by task B.
- A task may delete a segment that is being swapped out. If the swap-space release procedure does not allow for this case, the segment's swap space may be released and reallocated while the out-swapper is writing to it.

The in-swapper, out-swapper, and swap-space release procedures can control these interactions by maintaining a shared table of all segments that are in transition. They must use a semaphore or region to coordinate access to the shared table. The swap-space address can serve as the segment identifier in the table.

A virtual memory system gives each task the opportunity to affect other tasks in the system. Tasks compete with one another for real memory resources. One task may attempt to use memory in such a way as to unduly impede the progress of other tasks. The operating system must enforce policies which ensure that every task makes appropriate progress.

Virtual memory management policies are not unique to the iAPX 286; the computer-science literature contains many discussions of policies that apply to various classes of applications. There is, however, a distinction between *segmented* architectures and *paged* architectures. The iAPX 286 has a segmented architecture. Literature that deals with paged architectures may not apply to the iAPX 286. Refer to the paper by Denning (see "External Literature" in the Preface) for a broad survey of the science of virtual-memory management.

The remainder of this chapter is an introduction to memory-management policy. The policies you need to consider are of three types: fetch policy, placement policy, and replacement policy.

Fetch

The fetch policy determines which segment to bring from swap space into RAM and determines when to bring it in.

The simplest fetch policy is to bring in a segment *on demand*, that is, at the time it is referenced. Under this policy, the operating system brings in a segment from swap space only when the processor causes a not-present exception as the result of a reference to the segment.

All other fetch policies are in some way *anticipatory*. A simple example of an anticipatory fetch policy on the iAPX 286 is to always bring in the LDT of a task when bringing in its TSS. Some time-sharing systems implement an anticipatory policy that brings in all the segments of a task at once. This policy may be suitable for tasks that consist only of one code segment, one or two data segments, and stack segment (as, for example, simple BASIC programs submitted by students in a university environment). The swapping managers can use the segment register fields in the TSS (CS, DS, ES, SS) to identify the task's working set.

Attempting to implement such a full-task swap policy for more complex tasks that use many segments may result in frequently fetching segments that are not referenced in any one time slice.

The additional complexity of implementing an anticipatory fetch policy is justifiable only if the anticipatory policy performs better than the demand policy. Given the efficiency of the iAPX 286 exception mechanism and given that in a multitasking environment there is usually some other task to service while one task waits for the in-swapper to fetch a segment, an anticipatory policy typically does not provide significantly greater throughput. An anticipatory policy may give better performance, however, when judged by performance standards other than throughput (for example, interrupt latency for specific tasks).

Placement

Determining where in RAM to place an incoming segment is the subject of Chapter 3. In a virtual-memory system, however, the constant reallocation of real memory to segments of varying length places special demands on the operating system's memory management module. When choosing a space-management algorithm for a system that includes virtual memory, you may wish to give extra consideration to the trade-off between speed and memory fragmentation.

Replacement

The operating system may need to replace one or more other segments that already reside in real memory to make space for a segment that is the object of a "segment-not-present" fault. The replacement policy determines which segments to eliminate and when to eliminate them.

The choice of which segments to evict from RAM is crucial to the performance of the system. The goal is to evict only segments that will not soon be referenced. The difficulty is knowing which segments will not soon be referenced. Many different policies are discussed in the literature. They fall into three general classes:

- *Naive* policies that determine which segment to evict without any knowledge of segment access patterns
- *Historical* policies that use information about prior access patterns to determine the probability that a segment will soon be referenced
- *Optimal* policies that use analyses of actual program control flow to determine the probability that a segment will soon be referenced

HISTORICAL POLICIES

The iAPX 286 architecture supports gathering of historical information about actual segment access patterns via the accessed bit in executable and data segment descriptors. An operating-system module that periodically tests and resets the accessed bits of descriptors can develop a "profile" of segment usage. The information gathered this way can be used both by the replacement policy module for run-time decision making and by the operating system designers for improving replacement policy.

A "profiler" works best if it takes samples at regular intervals. To find all descriptors, it must step thru LDTs. A profiler does not need to examine all descriptors in the system each time it is invoked; it needs only to examine those of tasks that have run since the last time it was invoked. The same timer interrupt procedure used by the scheduler can invoke the profiler at appropriate times.

A profiler may run either as a separate task or as a shared procedure in the current task. A profiler that runs as a procedure in the current task can easily locate the LDT with an SLDT instruction. The LSL (load segment limit) instruction helps the profiler find the size of an LDT. The LAR (load access rights) instruction enables the profiler to test the accessed bit. A profiler that runs as a separate task may require the support of a PL-0 procedure that locates LDTs and tests accessed bits in other tasks.

A profiler must give special consideration to aliases. If the accessed bit in any of the aliases of a segment is set, the segment has been accessed. Here again, a PL-0 segment may be needed to step through the kernel's alias lists.

OPTIMAL POLICIES

Many of the optimal policies discussed in the literature are of theoretical interest only. They are used as a standard against which to measure the performance of more practical policies.

The segment-register fields of the TSS provide support for a trivial, but possibly useful, optimal policy. The replacement policy can assume that, next time any given task runs, all the segments identified by the CS, DS, SS, and ES fields of the TSS will be referenced. The processor loads all these registers during a task switch and causes a fault at that time for each not-present segment. Since the task cannot run if any one of these segments is not present, the replacement policy may as well replace all of them

at once. It then becomes possible to allocate swap space for these segments in such a way as to minimize seek time. Such a policy works well with the full-task fetch policy outlined previously.

THRASHING

If not carefully controlled, the I/O traffic in support of virtual memory may degrade system performance beyond acceptable limits. The worst case of performance degradation is called *thrashing*. This happens when RAM is committed to simulating an excessively large virtual-memory space and the behavior of the tasks in that space is such that no task can run without causing a not-present fault.

You can avoid thrashing by measuring or estimating the minimum amount of RAM a task needs in which to operate without causing frequent not-present faults. If the task loader knows this figure, it can refuse to load a task when not enough RAM is available. You can measure a task's RAM requirements with a profiler such as that described previously.

System Initialization 10

10 System Initialization

SYSTEM INITIALIZATION

The initialization performed at power-on or RESET by the 80286 processor is not, by itself, adequate for running in protected mode. Software must perform additional initialization before it is possible to fully use protected mode.

INITIAL STATE

When you power up an 80286 system or perform a RESET, the state of the processor is as follows:

- The MSW (machine status word) is zero; i.e., the 80286 starts running in the real-address mode.
- CS:IP be set to F000:FFF0, and the CS limit is 0FFFFH.
- The four high-order address lines A_{23-20} are automatically asserted for all subsequent references to CS until your initialization code changes CS.
- SS, DS, ES are set to zero, and the corresponding limit registers are set to 0FFFFH.
- The flag word is zero. This means that the 80286 starts running with the maskable interrupts disabled.

The initial state of the address lines and CS:IP causes the processor to begin executing instructions at physical address 0FFFFF0H. Presumably, this addresses a JMP instruction in an initialization procedure located in ROM or in RAM that has been loaded by another processor. The initialization procedure may occupy any portion of the last 65,536 bytes of the 16-megabyte address space. The JMP instruction at physical address 0FFFFF0H transfers control to the actual beginning of the initialization procedure. The first instruction that modifies the CS register causes the processor to cease asserting the high-order four address lines; therefore, the initialization procedure must avoid using any instructions that modify the CS register, except for the final JMP instruction.

The initial state of the DS, ES, and SS registers gives the initialization procedure access to the first 65,536 bytes of the address space. The initialization procedure may change these registers, however, to gain access to any location in the first megabyte of the address space. (With regard to segmentation and addressing, the 80286 in real-address mode behaves just as an 8086.) Access to other portions of memory is possible only after switching into protected mode.

SWITCHING TO PROTECTED MODE

You use the LMSW (load machine status word) instruction to set the PE bit in the MSW, thereby switching the 80286 into protected, virtual-address mode. The current privilege level (CPL) starts at zero. The segment registers continue to point to the same physical memory areas as in real-address mode.

Immediately after setting the PE flag, the initialization code must flush the processor's instruction queues by executing a JMP instruction. The 80286 fetches and decodes instructions and addresses before they are used; however, after a change into protected, virtual-address mode, the prefetched instruction information (which pertains to real-address mode) is no longer valid. A JMP forces the processor to discard the invalid information. An intrasegment JMP will cause the processor to drop the four high-order address lines; however, in protected mode this is not a problem. All addressing in protected mode uses the 24-bit base address from the segment descriptor; so, once in protected mode, all of physical memory is accessible.

You can do most of the initialization needed for protected mode either before or after switching into protected mode. If done after, however, you must be careful to order the code so that it does not use protected mode features that require initialization that is not yet completed.

Interrupt Vector

The initial state of the 80286 leaves interrupts disabled; however, to ensure a predictable action in case an exception or non-maskable interrupt (NMI) occurs, it is a good idea to initialize the IDT register. Since it is unlikely that the NMI interrupt handler or any exception handler has been initialized, the most appropriate value to load into the IDT register is zeros, thereby guaranteeing a shutdown should an interrupt happen. (The 80286 signals shutdown externally as an indication of a severe problem.) Later, when interrupt service routines are ready, you can change the IDT register to point to an actual IDT that contains gate descriptors for the interrupt routines. Interrupts may be enabled at that time.

Stack

Before you perform any stack operations, whether in real-address mode or in protected, virtual-address mode, you must load the SS register with a descriptor to a stack segment in RAM. If the SS register is loaded in real-address mode, it continues to point to the same segment after the switch into protected, virtual-address mode.

Global Descriptor Table

Before you change any segment register in protected, virtual-address mode, the GDT register must point to a valid GDT.

The GDT (as well as LDTs) should reside in RAM because the processor modifies the accessed bit of descriptors.

To allow full 16-megabyte addressing in the initialization code, you may find it convenient to build a temporary GDT that contains only enough descriptors to permit the initialization procedure to read the GDT segment from ROM or from a bootloadable module. After placing the real GDT into RAM, you can change the GDT register.

STARTING FIRST TASK

The initialization procedure can run awhile in protected mode without initializing the task register; however, before the first task switch, two conditions must prevail:

- There must be a valid task state segment (TSS) for the new task. The register fields of the TSS should have appropriate values, the segment register fields must point to valid segments or be null, the stack pointers for privilege levels numerically less than or equal to the initial CPL must point to valid stack segments, the LDT pointer must point to the GDT entry for a valid LDT (or be null if the task does not use an LDT) and, just as insurance, the back link of the TSS should be null.
- The task register must point to an area in which to save the current task state. After the first task switch, the information dumped in this area is not needed, and the area can be used for other purposes.

Starting the first task is simply a matter of executing a long JMP via the descriptor of a TSS or via a task gate to that descriptor. This task can perform any remaining initialization work while enjoying the full protection and virtual-address features of the iAPX 286, the state assumed by the development tools.

EXAMPLE OF INITIALIZATION

The following figures present a detailed example of one way to initialize a protected, virtual-address mode system. This example includes assembly language modules that work in conjunction with Builder specifications.

Figure 10-1 shows the primary initialization module ENTP (ENTer Protected mode). This module puts the 80286 into protected, virtual-address mode and invokes the first task constructed by BLD286. You use a module such as ENTP by binding it with other modules (presumably those constituting the operating system and the initial task) that you intend to place in EPROM.

The module SEGS shown in figure 10-2 supplies dummy segment descriptions for ENTP. The program INIT shown in figure 10-3 serves as the initial task. It merely displays a message when initialization is complete.

Initialization Module ENTP

The steps that ENTP takes are to

- Switch into protected mode
- Create a temporary GDT
- Create an IDT and GDT copy in RAM from tables in EPROM
- Point the CPU to the RAM tables
- Copy all the TSSs and LDTs identified in the GDT from EPROM to RAM
- Update the RAM GDT to point to the RAM copies
- JMP to the initialization task in the GDT

The initializations immediately following RESET_STARTUP are redundant. They simulate the hardware RESET initializations in case of a software reset or in case of a branch to RESET_STARTUP during debugging.

INITIAL_GDT is a temporary GDT containing descriptors for the IDT and the main GDT in EPROM (the one constructed by BLD286). Since the symbols for these descriptors, GDT_DESC and IDT_DESC, are PUBLIC, the Builder can insert the actual base and limit values into the table.

The code in segment ENTP_CODE is self-relocatable. In an EPROM-based system, you would specify to the Builder the actual address of the segment ENTP_CODE, making sure that the entry point resides at physical address FFFFF0H.

ENTP assumes a specific format for the GDT constructed by BLD286. The first two descriptors are the data-segment aliases for the GDT and the IDT. The remaining descriptors are grouped according to the template defined by TASK_ENTRY. Three adjacent descriptors identify the key segments of each task. ENTP adapts at run time to the actual number of such groups in the GDT. The task that ENTP initiates is identified by a fixed position in the GDT.

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                    date        PAGE    1


system-ID  iAPX286 MACRO ASSEMBLER Vx.y ASSEMBLY OF MODULE ENTP
OBJECT MODULE PLACED IN :F1:ENTP.OBJ
ASSEMBLER INVOKED BY:  ASM286.86 :F1:ENTP.A86


LOC  OBJ                  LINE     SOURCE

                             1 +1  $TITLE('Enter Protected Mode 960-516')
                             2
                             3                   NAME    ENTP
                             4                   PUBLIC  IDT_DESC,GDT_DESC,START
                             5     ;
                             6     ;       Switch the 80286 from real address mode into protected mode.
                             7     ;     The initial EPROM GDT, IDT, TSS, and LDT (if any) constructed by BLD286
                             8     ;     are copied from EPROM into RAM.  The RAM areas are defined by data
                             9     ;     segments allocated as fixed entries in the GDT.  The CPU registers for
                            10     ;     the GDT, IDT, TSS, and LDT are set to point at the RAM based
                            11     ;     segments.  The base fields in the RAM GDT are also updated to
                            12     ;     point at the RAM based segments.
                            13     ;
                            14     ;       Interrupts are disabled during this mode switching code.
                            15     ;     The EPROM based GDT, IDT, TSS, and LDT are checked to assure
                            16     ;     they are valid before copying them to RAM.  If any of the RAM-based
                            17     ;     alias segments are smaller than the EPROM segments they are to hold,
                            18     ;     halt or shutdown occurs.  In general any exception or NMI causes
                            19     ;     shutdown to occur until the first task is invoked.
                            20     ;
                            21     ;       If the RAM segment is larger than the EPROM segment, the RAM segment
                            22     ;     is expanded with zeroes.  If the initial TSS specifies an LDT,
                            23     ;     the LDT is also copied into LDT_ALIAS with zero fill if needed.
                            24     ;     The EPROM or RAM based GDT, IDT, TSS, and LDT segments may be located
                            25     ;     anywhere in physical memory.
                            26     ;
                            27 +1  $EJECT
                            28     ;
                            29     ;       Define layout of a descriptor.
                            30     ;
----                        31     DESC            STRUC
0000                        32     LIMIT           DW      0               ; Offset of last byte in segment
0002                        33     BASE_LOW        DW      0               ; Low 16 bits of 14-bit address
0004                        34     BASE_HIGH       DB      0               ; High 8 bits of 24-bit address
0005                        35     ACCESS          DB      0               ; Access rights byte
0006                        36     RES             DW      0               ; Reserved word
----                        37     DESC            ENDS
                            38     ;
                            39     ;       Define the fixed GDT selector values for the descriptors that
                            40     ;     define the EPROM based tables.
```

Figure 10-1. Initialization Module ENTP

121960-001

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                    date        PAGE    2

LOC  OBJ                  LINE     SOURCE

                           41      ;
  0008                     42      GDT_ALIAS       EQU     1*SIZE DESC  ; GDT(1) is data segment space for GDT
  0010                     43      IDT_ALIAS       EQU     2*SIZE DESC  ; GDT(2) is data segment space for IDT
  0018                     44      START_TSS_ALIAS EQU     3*SIZE DESC  ; GDT(3) is data segment space for TSS
  0020                     45      START_TASK      EQU     4*SIZE DESC  ; GDT(4) is TSS for starting task
  0028                     46      START_LDT_ALIAS EQU     5*SIZE DESC  ; GDT(5) is data segment space for LDT
                           47      ;
                           48      ;       Define machine status word bit positions.
                           49      ;
  0001                     50      PE              EQU     1            ; Protection enable
                           51      ;
                           52      ;       Define particular values of descriptor access rights byte.
                           53      ;
  0082                     54      DT_ACCESS       EQU     82H          ; Access byte value for an LDT
  0092                     55      DS_ACCESS       EQU     92H          ; Access byte value for data segment:
                           56                                           ;  expand up, level 0, writeable
  0081                     57      TSS_ACCESS      EQU     81H          ; Access byte value for an idle TSS
  0060                     58      DPL             EQU     60H          ; Privilege level field of access rights
  0001                     59      ACCESSED        EQU     1            ; Define accessed bit
  0004                     60      TI              EQU     4            ; Position of TI bit
  002C                     61      TSS_SIZE        EQU     44           ; Size of a TSS
  002A                     62      LDT_OFFSET      EQU     42           ; Position of LDT in TSS
  0007                     63      TIRPL_MASK      EQU     SIZE DESC-1  ; TI and RPL field mask
                           64 +1   $EJECT
                           65      ;
                           66      ;       Pass control from the power up address to the mode switch code.
                           67      ;     The segment containing this code must be at physical address FFFE10H
                           68      ;     to place the JMP instruction at physical address FFFFF0H.  The base
                           69      ;     address is chosen according to the size of this segment.
                           70      ;
----                       71      ENTP_CODE       SEGMENT ER
                           72
  FE10                     73      CS_OFFSET       EQU     0FE10H       ; Low 16 bits of starting address
01E0                       74                      ORG     0FFF0H-CS_OFFSET; Start at address FFFFF0H
01E0 E969FE                75                      JMP     RESET_STARTUP; Do not change CS!
                           76      ;
                           77      ;       Define the template for a temporary GDT used to locate the initial
                           78      ;     GDT and stack.  This data is copied to location 0.
                           79      ;     This space is also used for a temporary stack and finally serves
                           80      ;     as the TSS written into when entering the initial TSS.
                           91      ;
0000                       82                      ORG     0            ; Place remaining code below power_up
                           83                                           ;   address
0000                       84      INITIAL_GDT     LABEL   WORD
0000 0000                  85      NULL_DESC       DESC    <>           ; Filler and null IDT descriptor
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                    date        PAGE    3

LOC  OBJ                  LINE     SOURCE

0002 0000
0004 00
0005 00
0006 0000
0008 0000                   86     GDT_DESC        DESC    <>              ; Descriptor for EPROM GDT
000A 0000
000C 00
000D 00
000E 0000
0010 0000                   87     IDT_DESC        DESC    <>              ; Descriptor for EPROM IDT
0012 0000
0014 00
0015 00
0016 0000
0018 0000                   88     TEMP_DESC       DESC    <>              ; Temporary descriptor
001A 0000
001C 00
001D 00
001E 0000
                            89     ;
                            90     ;       Define a descriptor which points the GDT at location 0.
                            91     ;     This descriptor is also loaded into SS to define the initial
                            92     ;     protected mode stack segment.
                            93     ;
0020 3F00                   94     TEMP_STACK      DESC    <END_GDT-INITIAL_GDT-1,0,0,DS_ACCESS,0>
0022 0000
0024 00
0025 92
0026 0000
                            95 +1  $EJECT
                            96     ;
                            97     ;       Define the TSS descriptor used to allow the task switch to the
                            98     ;     first task to overwrite this region of memory.  The TSS overlays
                            99     ;     the initial GDT and stack at location 0.
                           100     ;
0028 3F00                  101     SAVE_TSS        DESC    <END_GDT-INITIAL_GDT-1,0,0,TSS_ACCESS,0>
002A 0000
002C 00
002D 81
002E 0000
                           102     ;
                           103     ;       Define the initial stack space and filler for the end of the TSS.
                           104     ;
0030 (8                    105                     DW      8 DUP (0)
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

121960-001

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                                  date          PAGE    4

LOC  OBJ                  LINE     SOURCE

     0000
     )
0040                      106      END_GDT         LABEL   WORD
                          107
0040                      108      START_POINTER   LABEL   DWORD
0040 0000                 109                      DW      0,START_TASK     ; Pointer to initial task
0042 2000
                          110      ;
                          111      ;       Define template for the task definition list.
                          112      ;
----                      113      TASK_ENTRY      STRUC                    ; Define layout of task description
0000                      114      TSS_SEL         DW      ?                ; Selector for TSS
0002                      115      TSS_ALIAS       DW      ?                ; Data segment alias for TSS
0004                      116      LDT_ALIAS       DW      ?                ; Data segment alias for LDT if any
----                      117      TASK_ENTRY      ENDS
                          118
0044 2000                 119      TASK_LIST       TASK_ENTRY      <START_TASK,START_TSS_ALIAS,START_LDT_ALIAS>
0046 1800
0048 2800
004A 0000                 120                      DW      0                ; Terminate list
                          121
004C                      122      RESET_STARTUP:
004C FA                   123              CLI                      ; No interrupts allowed!
004D FC                   124              CLD                      ; Use autoincrement mode
004E 33FF                 125              XOR     DI,DI            ; Point ES:DI at physical address 000000H
0050 8EDF                 126              MOV     DS,DI
0052 8EC7                 127              MOV     ES,DI
0054 8ED7                 128              MOV     SS,DI            ; Set stack at end of reserved area
0056 BC4000               129              MOV     SP,END_GDT-INITIAL_GDT
                          130 +1   $EJECT
                          131      ;
                          132      ;       Form an adjustment factor from the real CS base of FF0000H to the
                          133      ;     segment base address assumed by ASM286.  Any data reference made
                          134      ;     into CS must add an indexing term [BP] to compensate for the difference
                          135      ;     between the offset generated by ASM286 and the offset required from
                          136      ;     the base of FF0000H.
                          137      ;
0059                      138      START   PROC                     ; The value of IP at run time is not
                          139                                       ;   the same as used by ASM286!
0059 E80000               140              CALL    START1           ; Get true offset of START1
005C                      141      START1:
005C 5D                   142              POP     BP
005D 83ED5C               143              SUB     BP,OFFSET START1         ; Subtract ASM286 offset of START1
                          144                                       ;   leaving adjustment factor in BP
0060 2E0F015E00           145              LIDT    NULL_DESC[BP]            ; Set up null IDT to force shutdown
                          146                                       ;  on any protection error or interrupt
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                    date        PAGE    5

LOC  OBJ                  LINE     SOURCE

                          147      ;
                          148      ;Copy the EPROM-based temporary GDT into RAM.
                          149      ;
0065 8D7600               150              LEA     SI,INITIAL_GDT[BP]          ; Setup pointer to GDT
0068 B92000               151              MOV     CX,(END_GDT-INITIAL_GDT)/2  ; Set length
006B F3                   152       REP    MOVS    ES:WORD PTR [DI],CS:[SI]    ; Put into reserved RAM
006C 2EA5
                          153      ;
                          154      ;       Switch to protected mode and set up a stack, GDT, and LDT.
                          155      ;
006E 0F01E0               156              SMSW    AX                          ; Get current MSW
0071 0D0100               157              OR      AX,PE                       ; Set PE bit
0074 0F01F0               158              LMSW    AX                          ; Enter protected mode!
0077 EB00                 159              JMP     $+2                         ; Clear queue of instructions decoded
                          160                                                  ;  while in Real Address Mode
                          161                                                  ; CPL is now 0, CS still points at
                          162                                                  ;  FFFE10 in physical memory
                          163
0079 2E0F015620           164              LGDT    TEMP_STACK[BP]              ; Put initial GDT into RAM area
007E B82000               165              MOV     AX,TEMP_STACK-INITIAL_GDT   ; Setup SS with valid protected mode
0081 8ED0                 166              MOV     SS,AX                       ;  selector to the RAM GDT and stack
0083 33C0                 167              XOR     AX,AX                       ; Set the current LDT to null
0085 0F00D0               168              LLDT    AX                          ; Any references to it causes
                          169                                                  ;  an exception causing shutdown
0088 B82800               170              MOV     AX,SAVE_TSS-INITIAL_GDT     ; Set initial TSS into the low RAM
0088 0F00D8               171              LTR     AX                          ; The task switch needs a valid TSS
                          172 +1   $EJECT
                          173      ;
                          174      ;       Copy the EPROM based GDT into the RAM data segment alias.
                          175      ;     First the descriptor for the RAM data segment must be copied into
                          176      ;     the temporary GDT.
                          177      ;
008E 2E8B4608             178              MOV     AX,GDT_DESC[BP].LIMIT       ; Get size of GDT
0092 3D2F00               179              CMP     AX,6*SIZE DESC-1            ; Be sure the last entry expected by
                          180                                                  ;  this code is inside the GDT
0095 723C                 181              JB      BAD_GDT                     ; Jump if GDT isn't big enough
                          182
0097 BB0800               183              MOV     BX,GDT_DESC-INITIAL_GDT     ; Form selector to EPROM GDT
009A BE0800               184              MOV     SI,GDT_ALIAS                ; Get selector of GDT alias
009D E8D600               185              CALL    COPY_EPROM_DT               ; Copy into EPROM
00A0 BE1000               186              MOV     SI,IDT_ALIAS                ; Get selector of IDT alias
00A3 BB1000               187              MOV     BX,IDT_DESC-INITIAL_GDT     ; Indicate EPROM IDT
00A6 E8CD00               188              CALL    COPY_EPROM_DT
00A9 B80800               189              MOV     AX,GDT_DESC-INITIAL_GDT     ; Set up addressing into EPROM GDT
00AC 8ED8                 190              MOV     DS,AX
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

121960-001

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-515                    date        PAGE    6

LOC  OBJ                  LINE     SOURCE

00AE 8B0800               191              MOV     BX,GDT_ALIAS            ; Get GDT alias data segment selector
00B1 0F0117               192              LGDT    [BX]                    ; Set GDT to RAM GDT
                          193                                              ; SS and TR remain in low RAM
                          194      ;
                          195      ;       Copy all task's TSS and LDT segments into RAM
                          196      ;
00B4 8D5E44               197              LEA     BX,TASK_LIST[BP]        ; Define list of tasks to set up
00B7                      198      COPY_TASK_LOOP:
00B7 E81B00               199              CALL    COPY_TASKS              ; Copy them into RAM
00BA 83C306               200              ADD     BX,SIZE TASK_ENTRY      ; Go to next entry
00BD 2E8B07               201              MOV     AX,CS:[BX].TSS_SEL      ; See if there is another entry
00C0 0BC0                 202              OR      AX,AX
00C2 75F3                 203              JNZ     COPY_TASK_LOOP
                          204      ;
                          205      ;       With TSS, GDT, and LDT set, start up the initial task!
                          206      ;
00C4 8B0800               207              MOV     BX,GDT_ALIAS            ; Point DS at GDT
00C7 8ED8                 208              MOV     DS,BX
00C9 8B1000               209              MOV     BX,IDT_ALIAS            ; Get IDT alias data segment selector
00CC 0F011F               210              LIDT    [BX]                    ; Set IDT for errors and interrupts
00CF 2EFF6E40             211              JMP     START_POINTER[BP]       ; Start the first task!
                          212                                              ; The low RAM area is overwritten with
                          213                                              ;   the current CPU context
00D3                      214      BAD_GDT:
00D3 F4                   215              HLT                             ; Halt here if GDT isn't big enough
                          216
                          217      START   ENDP
                          218 +1   $EJECT
                          219      ;
                          220      ;       Copy the TSS and LDT for the task pointed at by CS:BX.
                          221      ;     If the task has an LDT it will be copied down, too.
                          222      ;     BX and BP are transparent.
                          223      ;
00D4                      224      BAD_TSS:
00D4 F4                   225              HLT                             ; Halt here if TSS is invalid
00D5                      226      COPY_TASKS      PROC
                          227
00D5 8E0800               228              MOV     SI,GDT_ALIAS            ; Get addressability to GDT
00D8 8EDE                 229              MOV     DS,SI
00DA 2E8B7702             230              MOV     SI,CS:[BX].TSS_ALIAS    ; Get selector for TSS alias
00DE 8EC6                 231              MOV     ES,SI                   ; Point ES at alias data segment
00E0 0F03C6               232              LSL     AX,SI                   ; Get length of TSS alias
00E3 2E8B37               233              MOV     SI,CS:[BX].TSS_SEL      ; Get TSS selector
00E6 0F02D6               234              LAR     DX,SI                   ; Get alias access rights
00E9 75E9                 235              JNZ     BAD_TSS                 ; Jump if invalid reference
                          236
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                    date        PAGE    7

LOC  OBJ                LINE     SOURCE

00EB 8AD6                237              MOV     DL,DH                   ; Save TSS descriptor access byte
00ED 80E69F              238              AND     DH,NOT DPL              ; Ignore privilege
00F0 80FE81              239              CMP     DH,TSS_ACCESS           ; See if TSS
00F3 75DF                240              JNZ     BAD_TSS                 ; Jump if not
                         241
00F5 0F03CE              242              LSL     CX,SI                   ; Get length of EPROM based TSS
00F8 83F92B              243              CMP     CX,TSS_SIZE-1           ; Verify it is of proper size
00FB 72D7                244              JB      BAD_TSS                 ; Jump if it is not big enough
                         245      ;
                         246      ;       Setup for moving the EPROM based TSS to RAM.
                         247      ;     DS points at GDT.
                         248      ;
00FD C6440592            249              MOV     [SI].ACCESS,DS_ACCESS   ; Make TSS into data segment
0101 8EDE                250              MOV     DS,SI                   ; Point DS at EPROM TSS
0103 E89B00              251              CALL    COPY_WITH_FILL          ; Copy DS segment to ES with zero fill
                         252                                              ;  CX has copy count, AX-CX fill count
                         253      ;
                         254      ;       Set the GDT TSS limit and base address to the RAM values.
                         255      ;
0106 B80800              256              MOV     AX,GDT_ALIAS            ; Restore GDT addressing
0109 8ED8                257              MOV     DS,AX
010B 8EC0                258              MOV     ES,AX
010D 2E8B3F              259              MOV     DI,CS:[BX].TSS_SEL      ; Get TSS selector
0110 2E8B7702            260              MOV     SI,CS:[BX].TSS_ALIAS    ; Get RAM alias selector
0114 A5                  261              MOVSW                           ; Copy limit
0115 A5                  262              MOVSW                           ; Copy low 16 bits of address
0116 AD                  263              LODSW                           ; Get high 8 bits of address
0117 8AE2                264              MOV     AH,DL                   ; Mark as TSS descriptor
0119 AB                  265              STOSW                           ; Fill in high address and access bytes
011A A5                  266              MOVSW                           ; Copy reserved word
                         267 +1   $EJECT
                         268      ;
                         269      ;       See if a valid LDT is specified for the startup task.
                         270      ;     If so then copy the EPROM version into the RAM alias.
                         271      ;
011B 2E8E5F02            272              MOV     DS,CS:[BX].TSS_ALIAS    ; Address TSS to get LDT
011F 8B362A00            273              MOV     SI,DS:WORD PTR LDT_OFFSET
0123 81E6F8FF            274              AND     SI,NOT TIRPL_MASK       ; Ignore TI and RPL
0127 7441                275              JZ      NO_LDT                  ; Skip this if no LDT used
                         276
0129 56                  277              PUSH    SI                      ; Save LDT selector
012A 0F02D6              278              LAR     DX,SI                   ; Test descriptor
012D 753C                279              JNZ     BAD_LDT                 ; Jump if invalid selector
                         280
012F 8AD6                281              MOV     DL,DH                   ; Save LDT descriptor access byte
0131 80E69F              282              AND     DH,NOT DPL              ; Ignore privilege
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                    date        PAGE    8


LOC  OBJ                  LINE     SOURCE

0134 80FE82                283              CMP     DH,DT_ACCESS            ; Be sure it is an LDT descriptor
0137 7532                  284              JNE     BAD_LDT                 ; Jump if invalid
                           285
0139 26C6440592            286              MOV     ES:[SI].ACCESS,DS_ACCESS; Mark LDT as data segment
013E 8EDE                  287              MOV     DS,SI                   ; Point DS at EPROM LDT
0140 0F03C6                288              LSL     AX,SI                   ; Get LDT limit
0143 E82600                289              CALL    TEST_DT_LIMIT           ; Verify it is valid
0146 8BC8                  290              MOV     CX,AX                   ; Save for later
                           291      ;
                           292      ;       Examine the LDT alias segment and, if good, copy to RAM.
                           293      ;
0148 2E8B7704              294              MOV     SI,CS:[BX].LDT_ALIAS    ; Get LDT alias selector
014C 8EC6                  295              MOV     ES,SI                   ; Point ES at alias segment
014E 0F03C6                296              LSL     AX,SI                   ; Get length of alias segment
0151 E81800                297              CALL    TEST_DT_LIMIT           ; Verify it is valid
0154 E84A00                298              CALL    COPY_WITH_FILL          ; Copy LDT into RAM alias segment
                           299      ;
                           300      ;       Set the LDT limit and base address to the RAM copy of the LDT.
                           301      ;
0157 2E8B7704              302              MOV     SI,CS:[BX].LDT_ALIAS    ; Restore LDT alias selector
015B 5F                    303              POP     DI                      ; Restore LDT selector
015C 880800                304              MOV     AX,GDT_ALIAS            ; Restore GDT addressing
015F 8ED8                  305              MOV     DS,AX
0161 8EC0                  306              MOV     ES,AX
0163 A5                    307              MOVSW                           ; Move the RAM LDT limit
0164 A5                    308              MOVSW                           ; Move the low 16 bits across
0165 AD                    309              LODSW                           ; Get the high 8 bits
0166 8AE2                  310              MOV     AH,DL                   ; Mark as LDT descriptor
0168 AB                    311              STOSW                           ; Set high address and access rights
0169 A5                    312              MOVSW                           ; Copy reserved word
016A                       313      NO_LDT:
016A C3                    314              RET                             ; All done
016B                       315      BAD_LDT:
016B F4                    316              HLT                             ; Halt here if LDT is invalid
                           317
                           318      COPY_TASKS      ENDP
                           319 +1   $EJECT
                           320      ;
                           321      ;       Test the descriptor table size in AX to verify that it is an
                           322      ;     even number of descriptors in length.
                           323      ;
016C                       324      TEST_DT_LIMIT   PROC
                           325
016C 50                    326              PUSH    AX                      ; Save length
016D 2407                  327              AND     AL,7                    ; Look at low order bits
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

121960-001

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                          date        PAGE    9

LOC  OBJ                  LINE     SOURCE

016F 3C07                 328              CMP     AL,7                    ; Must be all ones
0171 58                   329              POP     AX                      ; Restore length
0172 7501                 330              JNE     BAD_DT_LIMIT
                          331
0174 C3                   332              RET                             ; All OK
0175                      333      BAD_DT_LIMIT:
0175 F4                   334              HLT                             ; Die!
                          335
                          336      TEST_DT_LIMIT   ENDP
                          337      ;
                          338      ;       Copy the EPROM DT at selector BX in the temporary GDT to the alias
                          339      ;     data segment at selector SI.  Any improper descriptors or limits
                          340      ;     cause shutdown!
                          341      ;
0176                      342      COPY_EPROM_DT   PROC
                          343
0176 8CD0                 344              MOV     AX,SS                   ; Point ES:DI at temporary descriptor
0178 8EC0                 345              MOV     ES,AX
017A 26C6470592           346              MOV     ES:[BX].ACCESS,DS_ACCESS; Mark descriptor as a data segment
017F 26C747060000         347              MOV     ES:[BX].RES,0           ; Clear reserved word
0185 0F03C3               348              LSL     AX,BX                   ; Get limit of EPROM DT
0188 8BC8                 349              MOV     CX,AX                   ; Save for later
018A E8DFFF               350              CALL    TEST_DT_LIMIT           ; Verify it is a proper limit
018D BF0800               351              MOV     DI,GDT_DESC-INITIAL_GDT ; Address EPROM GDT in DS
0190 8EDF                 352              MOV     DS,DI
0192 BF1800               353              MOV     DI,TEMP_DESC-INITIAL_GDT; Get selector for temporary descriptor
0195 57                   354              PUSH    DI                      ; Save offset for later use as selector
0196 AD                   355              LODSW                           ; Get alias segment size
0197 E8D2FF               356              CALL    TEST_DT_LIMIT           ; Verify it is an even multiple of
                          357                                              ;  descriptors in length
019A AB                   358              STOSW                           ; Put length into temporary
019B A5                   359              MOVSW                           ; Copy remaining entries into temporary
019C A5                   360              MOVSW
019D A5                   361              MOVSW
019E 07                   362              POP     ES                      ; ES now points at the GDT alias area
019F 8ED8                 363              MOV     DS,BX                   ; DS now points at EPROM DT as data
                          364                                              ; Copy segment to alias with zero fill
                          365                                              ; CX is copy count, AX-CX is fill count
                          366                                              ; Fall into COPY_WITH_FILL
                          367
                          368      COPY_EPROM_DT   ENDP
                          369 +1   $EJECT
                          370      ;
                          371      ;       Copy the segment at DS to the segment at ES for length CX.
                          372      ;     Fill the end with AX-CX zeroes.  Use word operations for speed but
                          373      ;     allow odd byte operations.
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

121960-001

```
iAPX286 MACRO ASSEMBLER    Enter Protected Mode 960-516                    date        PAGE   10

LOC  OBJ                  LINE     SOURCE


                          374      ;
01A1                      375      COPY_WITH_FILL  PROC
                          376
01A1 33F6                 377              XOR     SI,SI                   ; Start at beginning of segments
01A3 33FF                 378              XOR     DI,DI
01A5 2BC1                 379              SUB     AX,CX                   ; Form fill count
01A7 83C101               380              ADD     CX,1                    ; Convert limit to count
01AA D1D9                 381              RCR     CX,1                    ; Allow full 64K move
01AC F3                   382        REP   MOVSW                           ; Copy DT into alias area
01AD A5
01AE 91                   383              XCHG    AX,CX                   ; Get fill count and zero AX
01AF 7307                 384              JNC     EVEN_COPY               ; Jump if even byte count on copy
                          385
01B1 A4                   386              MOVSB                           ; Copy odd byte
01B2 08C9                 387              OR      CX,CX
01B4 7409                 388              JZ      EXIT_COPY               ; Exit if no fill
                          389
01B6 AA                   390              STOSB                           ; Even out the segment offset
01B7 49                   391              DEC     CX                      ; Adjust remaining fill count
01B8                      392      EVEN_COPY:
01B8 D1E9                 393              SHR     CX,1                    ; Form word count on fill
01BA F3                   394        REP   STOSW                           ; Clear unused words at end
01BB AB
01BC 7301                 395              JNC     EXIT_COPY               ; Exit if no odd byte remains
                          396
01BE AA                   397              STOSB                           ; Clear last odd byte
01BF                      398      EXIT_COPY:
01BF C3                   399              RET
                          400
                          401      COPY_WITH_FILL  ENDP
                          402
----                      403      ENTP_CODE       ENDS
*** WARNING #160, LINE #403, SEGMENT CONTAINS PRIVILEGED INSTRUCTIONS
                          404                      END

ASSEMBLY COMPLETE,   1 WARNING,   NO ERRORS
```

Figure 10-1. Initialization Module ENTP (Cont'd.)

```
iAPX286 MACRO ASSEMBLER    DEFINE SEGMENTS NEEDED FOR INIT CODE                    date        PAGE    1


system-ID  iAPX286 MACRO ASSEMBLER Vx.y ASSEMBLY OF MODULE SEGMENT_DEF
OBJECT MODULE PLACED IN :F1:SEGS.OBJ
ASSEMBLER INVOKED BY:  :F1:ASM286.86 :F1:SEGS.A86

LOC  OBJ                  LINE     SOURCE

                             1 +1  $TITLE('DEFINE SEGMENTS NEEDED FOR INIT CODE')
                             2
                             3             NAME    SEGMENT_DEF
                             4
---- --                      5     INIT_STACK0     SEGMENT RW          ; Define stack so mode switch code can
0000 ????                    6                     DW      ?           ;  run in protected mode
---- --                      7     INIT_STACK0     ENDS
                             8
---- --                      9     LDT_SEG         SEGMENT RW          ; Define alias segments whose true
0000 (8                     10                     DW      8 DUP (?)   ;  size must be set by the BLD286
     ????                   )
---- --                     11     LDT_SEG         ENDS                ;  command file
                            12
---- --                     13     TSS_SEG         SEGMENT RW
0000 (8                     14                     DW      8 DUP (?)
     ????                   )
---- --                     15     TSS_SEG         ENDS
                            16
---- --                     17     IDT_SEG         SEGMENT RW
0000 (8                     18                     DW      8 DUP (?)
     ????                   )
---- --                     19     IDT_SEG         ENDS
                            20
---- --                     21     GDT_SEG         SEGMENT RW
0000 (8                     22                     DW      8 DUP (?)
     ????                   )
---- --                     23     GDT_SEG         ENDS
                            24
                            25             END

ASSEMBLY COMPLETE,  NO WARNINGS,  NO ERRORS
```

Figure 10-2. Dummy Segments for ENTP

121960-001

```
iAPX286 MACRO ASSEMBLER   INITIAL TASK                                    date        PAGE    1


system-ID  iAPX286 MACRO ASSEMBLER Vx.y ASSEMBLY OF MODULE INIT_MOD
OBJECT MODULE PLACED IN :F1:INIT.OBJ
ASSEMBLER INVOKED BY:  :F1:ASM286.86 :F1:INIT.A86


LOC  OBJ                  LINE     SOURCE

                             1 +1  $TITLE('INITIAL TASK')
                             2
                             3             NAME    INIT_MOD
                             4             EXTRN   CO:FAR
                             5
----                         6     INIT_STACK      STACKSEG 10
                             7
----                         8     INIT_DATA       SEGMENT RW PUBLIC
0000 ????                    9     DATA1           DW          ?
----                        10     INIT_DATA       ENDS
                            11
----                        12     INIT_CODE       SEGMENT ER
                            13                     ASSUME  DS:INIT_DATA
                            14
0000 496E697469616C         15     MESSAGE         DB      'Initialization Complete!',0DH,0AH,0
     697A6174696F6E
     20436F6D706C65
     746521
0018 0D
0019 0A
001A 00
                            16
001B                        17     INIT_START:
001B BE0000                 18             MOV     SI,OFFSET MESSAGE
001E FC                     19             CLD
001F                        20     MESS_LOOP:
001F 2EAC                   21             LODS    CS:BYTE PTR [SI]
0021 0AC0                   22             OR      AL,AL
0023 7408                   23             JZ      INIT_EXIT
                            24
0025 50                     25             PUSH    AX
0026 9A0000----      E      26             CALL    CO
002B EBF2                   27             JMP     MESS_LOOP
002D                        28     INIT_EXIT:
002D F4                     29             HLT
                            30
----                        31     INIT_CODE       ENDS
*** WARNING #160, LINE #31, SEGMENT CONTAINS PRIVILEGED INSTRUCTIONS
                            32                     END     INIT_START,DS:INIT_DATA,SS:INIT_STACK

ASSEMBLY COMPLETE,   1 WARNING,   NO ERRORS
```

Figure 10-3. Initial Task INIT

121960-001

```
                                                
```

Binding and Loading

11

CHAPTER 11
BINDING AND LOADING

Binding is the process of converting symbolic references into an implementation that the processor can utilize. The binding process spreads across many stages of software development, including source code, translators, binding utilities, loaders, and execution itself. Intel's program development tools include features that perform much of the needed binding. In some static systems no additional binding may be needed. In dynamic systems, however, you may choose to incorporate some binding functions in the operating system and related software in order to create a style of binding appropriate for the dynamic nature of your application.

BINDING MODEL

To ensure that the binding process works correctly, it is a good idea to start with a model of the system structure you wish to achieve. A binding model includes these factors:

- Modules—dividing programming into compilation units
- Segmentation—distributing instructions and data of modules among physical segments
- Interfaces—specifying the connections among modules
- Naming—choosing names for modules, segments, and interfaces, avoiding ambiguity but promoting flexibility
- Timing—determining when to bind various types of symbolic references

As an example of a binding model, assume that the example modules presented in prior chapters constitute a complete operating system, called XOS, and apply each of these factors to XOS.

Modules

The criteria used to separate functions into modules may include

- Comprehensibility. Each module collects together functions that support a single operating system concept of limited scope (for example, aliases, synchronization).
- Information hiding. Each module implements a data structure (for example, the alias lists in the ALIAS module) that you can manipulate only by calling the procedures defined in the module for that purpose. Other modules cannot access the data structure.
- Independent development. You can choose modules so that each can be developed by a different programmer. Points of cooperation and communication among programmers include only the specified interfaces among modules. Minimizing the number and complexity of interfaces reduces project administration costs.
- Structured testing. Testing of the whole system is simpler when you choose modules so that each can be tested by itself, before testing its interactions with other modules.
- Flexibility. When you can anticipate changes to the system, you can limit the effects of the changes on other modules by isolating the areas of expected change in a module.

Only the first of these criteria is significant to XOS, but all may be applicable to your operating system design.

Modules are relevant to binding in that they (indirectly) define the units that can be distributed among physical segments. A module is a single compilation unit. Intel's translators divide each compilation unit into *logical segments*. Each logical segment is a named object that can be assigned to a *physical segment*. You can use the development tool Binder to combine more that one logical segment into a physical segment.

Segmentation

Protection requirements partially dictate how to distribute modules among physical segments:

- Data and procedures of different privilege levels must reside in different physical segments.
- Data and procedures of one task must reside in different segments from those of other tasks unless sharing is explicitly intended.
- Instructions must reside in different physical segments from writable data items.
- Data structures for which you wish to provide individual protection (for example, the semaphores and mailboxes discussed in Chapter 5) must each be in a separate segment.

Operating system procedures that have the same privilege level and are shared via the GDT can be combined in just one segment (assuming that the total size does not exceed 64K bytes). In fact, doing so has two advantages: all intermodule calls can be implemented as short CALL instructions, avoiding the additional processing associated with changing the contents of the CS register, and more GDT slots are available for other purposes. Of the procedures in XOS, the level-zero procedures presented in Chapters 3 thru 5 can be combined in one segment, while the level-one I/O interface procedures can go in another segment. Each device-driver task has its own code, data, and stack segments.

Interfaces

Possible mechanisms for implementing the interfaces among modules are

- Intrasegment (short) references. References to data or procedures in the same segment are most efficiently implemented when you can use the current contents of a segment register.
- Intersegment (long) references. References to data or procedures in a segment not currently indicated by one of the segment registers must cause a segment selector to be loaded into a segment register. Intersegment references permit sharing of functions among many modules and permit access to functions at another privilege level.
- Sharing by value. Procedures and data can be shared by including a copy of the data or procedures in the same unit (segment or task, as appropriate) as each procedure that uses them. While this approach can yield more efficient execution in some cases, it has limited applicability. It is usually best to share dynamically changing data by name, so that all the procedures that use it can obtain the most up-to-date version. Sharing large or widely used data or procedures by name uses main memory more efficiently.
- Sharing by name. The "name" referred to here is the descriptor of the shared data or procedure. Chapter 5 discusses methods of sharing by name (namely GDT, common LDT, and aliases).

In XOS, all tasks share the segments of the kernel and I/O-interface segments via the GDT. Applications procedures use long calls to access the primitives in these segments. Calls within the kernel level or within the I/O level to private procedures at the same level are short calls.

Naming

The names by which you reference the components of a system influence the way the system can be bound. Classes of names over which you have some degree of control are

- Modules. Builder uses the name of a module to represent all publics or all segments within the module. Debuggers use module names to qualify identifiers of variables and procedures.
- Logical segments. The choice of logical segment names determines the possibilities for segment combination. Binder combines logical segments that have the same name as well as compatible combine-types.
- Physical segments. The names of physical segments give you the ability to assign segment attributes individually using the Builder.
- Publics. Public identifiers must be unique among the modules that are bound together.
- Gates. Gate names identify the entry points to procedures. You can use gate names to give entry points different public names than those used in the source language.

Timing

With respect to time, you can rate bindings on a scale running from early to late. An example of early binding is a compiler's assignment of segment-relative locations to procedures and data. The latest binding is that accomplished by the processor as it adds a segment-relative location to the 24-bit base address of the segment. Between these extremes are other opportunities for binding:

- Post-compile-time. Through Intel's utilities Builder and Binder, you can combine segments, resolve intersegment references, allocate descriptor table slots, and allocate memory.
- Load-time. The loader can incorporate various levels of binding.
  a. If the object module contains fix-up information (as in a linkable module), the loader can bind all references as it loads the segments of a task.
  b. The call gates of the iAPX 286 architecture make it possible for the loader to efficiently resolve references to predefined classes of procedures (to operating system primitives, for example) at the time a program is being loaded. This chapter presents an example of this form of load-time binding in a later section.
- Run-time. Call gates also permit binding to an executable segment at the time it is first referenced. By resetting the present bit in a call gate to an unloaded segment, you ensure that a trap will occur when the gate is used. The "not present" trap handler can then load the required segment, allocate a descriptor for it, and fix up the call gate.

IMPLEMENTING ACCORDING TO THE MODEL

With a binding model in mind, consider how to implement that model during the various stages of system development: in source code, by translators, by binding utilities, by loaders, by the operating system, and finally by the processor itself.

The model behind the example operating system, XOS, includes these components:

- Each module contains functionally related data structures and procedures.
- One segment contains level-zero (kernel) procedures, another segment contains level-zero data, another segment level-one (I/O) procedures, and another level-one data. Operating system tasks (such as device drivers) have their own distinct segments.
- The GDT contains descriptors of kernel and I/O segments, making them sharable by all tasks. Gates for operating-system primitives, however, reside in the LDTs of the tasks that use them.
- The segment names created by PL/M-286 set the standard for segment naming in general. (Refer to the chapter entitled "Linking to Modules Written in Other Languages" in the *PL/M-286 User's Guide* for definition of PL/M-286 segment names).
- Tasks will be loaded dynamically.
- Load-time binding to XOS primitives is an option. (This is one reason for placing gates for operating-system primitives in LDTs.)

Source Code

Since some of the logical segments declared in assembly language may be combined after assembly by the Builder, the assembler needs to know whether the object of an external reference will be in one of those segments whose descriptors it assumes to be loaded in segment registers. If the reference is to another segment, the assembler must emit instructions that change the contents of a segment register. The programmer supplies this information via additional assembly language syntax. A variety of forms are available for this purpose, such as

```
SEGMENT
ASSUME
NEAR and FAR variants of PROC and LABEL
segment overrides (for example ES:TABLE_ITEM)
```

(Refer to the *ASM286 Assembly Language Reference Manual* for details on the use of these items.)

The module DISP containing the dispatcher is the only kernel module of XOS written in assembly language (refer to Chapter 4). The logical code segment has the name NUCLEUS_CODE and combine type ER so that it will combine with PL/M-286 segments of the same name. This module has no logical data segment. The procedure DISPATCHER is a NEAR procedure because only other kernel procedures in the same physical segment call it.

Compilers

With PL/M-286, decisions regarding segmentation are not imbedded in source code but rather are implemented by the compiler according to compiler control statements that you supply. (Refer to the SMALL, COMPACT, LARGE, and extended segmentation controls in the *PL/M-286 User's Guide*.) With these control statements, you have nearly as much control over system structure as with assembly language, but changes in system structure do not require changes to source code. It is just as important, however, that use of these controls conform to a consistent model of system structure.

Figure 11-1 shows the segmentation controls used for compiling the kernel modules of XOS. These controls define a subsystem named NUCLEUS that contains all the PL-0 modules of XOS. The PL/M-286 compiler prefixes the names of the code and data segments with the subsystem name. The list of exports includes all the primitives. For each of the procedures named in the exports list, the

```
$ COMPACT (NUCLEUS HAS
$   SLOT                ,  POINT              ,
$   GATE                ,  MEMORY             ,
$   DISPATCHER          ,  ALIAS              ,
$   SEMAPH              ,  MAILBOX            ,
$   INTRUPT             ,  TASK               ,
$   DESCRIPTOR          ,  GATE               ,
$   DISQUE              ,  MESSAGE            ;
$ EXPORTS
$   RESERVE_SLOTS       ,  RELINQUISH_SLOTS   ,
$   ALLOCATE            ,  FREE_SEG           ,
$   CREATE_ALIAS        ,  CHANGE_AR          ,
$   WAIT_SEMAPHORE      ,  SIGNAL_SEMAPHORE   ,
$   SEND_MESSAGE        ,  RECEIVE_MESSAGE    ,
$   CREATE_LDT          ,  CREATE_TASK        ,
$   LOAD_LDT            ,  LOAD_LDT_GATE      ,
$   GET_GATE_POINTER    ,
$   ATTACH_TO_INTERRUPT ,  WAIT_FOR_INTERRUPT    )
```

Figure 11-1. Subsystem for Kernel Exports

compiler generates a long RET instruction at the end of the procedure or at RETURN statements. This enables procedures in other segments to call the the kernel procedures. The keyword COMPACT tells the compiler to generate short RET instructions for procedures not named in the export list.

Binding Utilities

Intel's iAPX 286 Binder (BND286) and System Builder (BLD286) provide a variety of binding services, including

- Combining logical segments that have the same name and combine type
- Resolving references among modules
- Constructing templates for GDT, LDT, and TSSs
- Allocating memory for bootloadable portions of the system
- Assigning access rights to segments

- Constructing gates
- Formatting object files for the convenience of boot loaders and dynamic loaders
- Creating export modules that contain gates for operating-system interfaces

Figure 11-2 shows the Binder specifications that combine the level-zero modules of XOS. The input modules contain only three unique segment names (the PL/M-286 names: NUCLEUS_CODE, NUCLEUS_DATA, and STACK); therefore, the output module contains just three segments: one for instructions, one for static data items, and one for the level-zero stack. The name of the output module is NUCLEUS; it is written to the file NUC.LNK. Similar specifications combine the level-one modules.

Figure 11-3 shows the specifications to build a bootloadable file for the example operating system. The SEGMENT statement assigns privilege levels to to each of the segments; segments not mentioned in this clause receive privilege level 3 (PL 3) by default.

The TABLE statement defines the descriptor tables. The RESERVE clause allocates space for the working descriptors used by such modules as the memory-management module described in Chapter 3. The ENTRY clause identifies the remaining segment descriptors that belong in each table. Builder allocates slots for each of these descriptors.

The TASK statement provides information for contructing TSSs. The identifier assigned to each task is the identifier of the descriptor of its TSS. The OBJECT clause identifies the module containing the information Builder can use to fill the segment-register and initial stack fields of the TSS.

The GATE statement creates gates for each of the public procedures that are XOS primitives, assigns a privilege level to each gate, and gives each a name different from the procedure name.

The EXPORT statement creates a linkable module KERNEL in file XOS.EXP that application modules can use for binding to the gates for XOS primitives.

```
RUN :F2:BND286 &
:F1:POINT.OBJ ,     :F1:SLOT.OBJ,   &
:F1:MEMORY.OBJ,     :F1:DISP.OBJ,   &
:F1:ALIAS.OBJ ,     :F1:SEMAPH.OBJ, &
:F1:MBOX.OBJ  ,     :F1:INTRPT.OBJ, &
:F1:DESCR.OBJ ,     :F1:DISQUE.OBJ, &
:F1:TASK.OBJ  ,     :F1:MESSAG.OBJ, &
PLM286.LIB                          &
NAME (NUCLEUS) OBJECT (:F1:NUC.LNK) NOLOAD DEBUG
```

Figure 11-2. Binder Specifications for XOS Kernel

```
system-ID  iAPX286 SYSTEM BUILDER, Vx.y

INPUT FILES:  :F1:NUC.LNK, :F1:URTASK.OBJ, :F1:CONSOL.OBJ, :F1:LOADER.LNK,
              :F2:LARGE.LB2, :F1:STACKS.OBJ
OUTPUT FILE:  :F1:XOS
CONTROLS SPECIFIED:  TITLE(Example O.S.) BOOTLOAD BUILDFILE(:F1:XOS.BLD)
                     OBJECT(:F1:XOS)

BUILD FILE: :F1:XOS.BLD

     1          EXAMPLE_OS;
     2
     3          SEGMENT
     4            NUCLEUS_CODE                (DPL=0),
     5            NUCLEUS_DATA                (DPL=0),
     6            NUCLEUS.STACK               (DPL=0, LIMIT=+20),
     7            URTASK_CODE                 (DPL=0),
     8            URTASK_DATA                 (DPL=0),
     9            URTASK.STACK                (DPL=0, LIMIT=+20H),
    10            LQ_PLM286_LIB_CODE          (DPL=0, CONFORMING),
    11            LARGE_V1P0.STACK0           (DPL=0),
    12            LARGE_V1P0.STACK1           (DPL=1),
    13            LARGE_V1P0.STACK2           (DPL=2),
    14            CONSOLE_DRIVER_CODE         (DPL=1),
    15            CONSOLE_DRIVER_DATA         (DPL=1),
    16            CONSOLE_DRIVER.STACK        (DPL=1, LIMIT=+20),
    17            CONSOLE_STACK0              (DPL=0, LIMIT=+20),
    18            LOADER_STACK0               (DPL=0, LIMIT=+20);
    19
    20          GATE
    21            XQ_ATTACH_TO_INTERRUPT (ENTRY=ATTACH_TO_INTERRUPT, DPL=1),
    22            XQ_WAIT_FOR_INTERRUPT  (ENTRY=WAIT_FOR_INTERRUPT,  DPL=1),
    23            XQ_RESERVE_SLOTS       (ENTRY=RESERVE_SLOTS,       DPL=3),
    24            XQ_RELINQUISH_SLOTS    (ENTRY=RELINQUISH_SLOTS,    DPL=3),
    25            XQ_ALLOCATE            (ENTRY=ALLOCATE,            DPL=3),
    26            XQ_FREE_SEG            (ENTRY=FREE_SEG,            DPL=3),
    27            XQ_CREATE_ALIAS        (ENTRY=CREATE_ALIAS,        DPL=3),
    28            XQ_CHANGE_AR           (ENTRY=CHANGE_AR,           DPL=3),
    29            XQ_WAIT_SEMAPHORE      (ENTRY=WAIT_SEMAPHORE,      DPL=3),
    30            XQ_SIGNAL_SEMAPHORE    (ENTRY=SIGNAL_SEMAPHORE,    DPL=3),
    31            XQ_SEND_MESSAGE        (ENTRY=SEND_MESSAGE,        DPL=3),
    32            XQ_RECEIVE_MESSAGE     (ENTRY=RECEIVE_MESSAGE,     DPL=3),
    33            XQ_CREATE_LDT          (ENTRY=CREATE_LDT,          DPL=3),
    34            XQ_CREATE_TASK         (ENTRY=CREATE_TASK,         DPL=3),
    35            XQ_LOAD_LDT            (ENTRY=LOAD_LDT,            DPL=3),
    36            XQ_LOAD_LDT_GATE       (ENTRY=LOAD_LDT_GATE,       DPL=3),
    37            XQ_GET_GATE_POINTER    (ENTRY=GET_GATE_POINTER,    DPL=3),
    38            TIME_SLICE (INTERRUPT, DPL=0, ENTRY=TIMINT);
    39
    40          TABLE
    41            GDT (RESERVE=(4..9), ENTRY=(
    42                      NUCLEUS_CODE,
    43                      NUCLEUS_DATA,
    44                      NUCLEUS.STACK,
    45                      LQ_PLM286_LIB_CODE,
    46                      LARGE_V1P0.STACK,
```

Figure 11-3. Builder Specifications for XOS

```
47                          LARGE_V1P0.STACK0,
48                          LARGE_V1P0.STACK1,
49                          LARGE_V1P0.STACK2
50                        )
51                ),
52              URTASK_LDT (ENTRY=(URTASK)),
53              CONSOLE_LDT (ENTRY=(CONSOLE_DRIVER, CONSOLE_STACK0)),
54              LOADER_TASK_LDT (ENTRY=(LOADER, LOADER_STACK0)),
55              IDT (ENTRY=(15:TIME_SLICE));
56
57            TASK
58              ADAM (INITIAL,  OBJECT = URTASK,
59                              LDT    = URTASK_LDT),
60              CONSOLE_DEVICE (OBJECT = CONSOLE_DRIVER,
61                              LDT    = CONSOLE_LDT,
62                              STACKS = (CONSOLE_STACK0)),
63              LOADER_TASK    (OBJECT = LOADER,
64                              LDT    = LOADER_TASK_LDT,
65                              STACKS = (LOADER_STACK0));
66
67              EXPORT #:F1:XOS.LB2 (KERNEL (
68                XQ_RESERVE_SLOTS,
69                XQ_RELINQUISH_SLOTS,
70                XQ_ALLOCATE,
71                XQ_FREE_SEG,
72                XQ_CREATE_ALIAS,
73                XQ_CHANGE_AR,
74                XQ_WAIT_SEMAPHORE,
75                XQ_SIGNAL_SEMAPHORE,
76                XQ_SEND_MESSAGE,
77                XQ_RECEIVE_MESSAGE));
78
79            END

BUILD FILE PROCESSING COMPLETED
```

Figure 11-3. Builder Specifications for XOS (Cont'd.)

These specifications do not illustrate all the features of Builder; refer to the *iAPX 286 System Builder User's Guide* for a complete description of Builder syntax.

OVERVIEW OF LOADING

The loader in a dynamic system is not only responsible for copying a program into main memory, but is also a step in the binding process. A loader installs the actual TSS and LDT for a task, thereby making it possible for the processor to interpret the task's memory references.

If all symbolic references are already resolved, a loader's work is simple. The iAPX 286 object module format (OMF) organizes segment information to facilitate rapid loading with little decision-making by the loader program.

Converting a Program Into a Task

Before an operating system can switch control to a new task, these structures must be initialized:

- The TSS (data the processor needs in order to run the task).
- The task database (TDB) (data the operating system needs in order to run the task).

Either the following structures must be present or some mechanism must be in place to make them present when necessary:

- Stack segments for each privilege level at which the task runs. (A stack-fault handler could automatically allocate a stack the first time it is used.)
- Executable and data segments. (A virtual memory system could make these present when referenced.)

Most programs also need an LDT to contain descriptors for segments private to the task. An LDT is not required, however, if all descriptors used by the task reside in the GDT.

The initial values in the CS:IP fields of the TSS should point to the entry point of the first procedure of the new task.

STYLES OF LOADERS

There are several ways to structure a loader. The following are just a few examples:

1. As a separate and permanent task. With such a structure the loader constructs data segments in the format of the new task's TSS and LDT. To create the new task, it passes descriptors for these segments to privilege-level 0 (PL-0) procedures that convert them to system segments and start the new task.
2. As procedures that run within an existing task. This approach suits the UNIX EXECUTE function, where the FORK function creates the task in which the loader runs as a duplicate of the task that called the FORK function. The loader deletes the descriptors it finds in the task's LDT (thereby deleting the associated segments unless aliases exist in other tasks) and creates new descriptors for the segments it loads.
3. As procedures in a skeletal task created by the operating-system kernel. The loader has only to allocate an LDT and install descriptors for the segments it loads.

For approaches 2 and 3, the procedures of the loader may have segment descriptors and gates in the GDT so as not to encumber the LDTs.

KERNEL SUPPORT

A loader normally runs at PLs 2 or 3 because it uses of both kernel-level procedures (for example, ALLOCATE to create a segment for the new task) and PL-1 procedures (for example, the I/O procedures to read object modules from disk). However, task creation involves some functions that only

PL-0 procedures can execute. For these functions the operating system must provide some additional support. These functions include

- Changing the access rights byte of a descriptor. The loader can create a writable data segment (using a level-0 procedure such as the ALLOCATE procedure described in Chapter 3) into which to read a segment from the object file of the program being loaded. But, if that segment is to have some other type in the new task (the segment might be an executable segment, for example), the access rights byte must be changed.
- Filling the new task's LDT with the base addresses of its segments. When the loader allocates a segment for the new task, the ALLOCATE procedure places the physical base address of the segment in a descriptor table (presumably the loader's LDT, but possibly the GDT). Only PL-0 procedures should have access to physical addresses in descriptor tables.
- Allocating the stack segment for PL 0. Insufficient space in this segment can cause failure of PL-0 procedures. Also, if the kernel requires the TSS to be located in the PL-0 stack segment (as suggested in Chapter 4), then the kernel should handle the complexities of setting up such a structure.
- Initializing the task database (TDB) for the new task. Only PL-0 procedures have access to this data structure. (Refer to Chapter 4 for a discussion of the task database.)
- Entering the new task in the scheduling queues and setting the back-link and nested task flag in the new TSS. These operations are crucial to proper functioning of the scheduler and therefore should be done at PL 0.

iAPX 286 Object Module Format

Intel has defined object module formats (OMFs) for the iAPX 286 to be used by translators, object-program utilities, and loaders. There are two classes of object module:

- Linkable modules, which are produced by translators and consumed by Builder and Binder. Binder can also produce a linkable module from one or more input linkable modules in a process known as *incremental binding*.
- Loadable modules, which are produced by the Binder and the Builder and consumed by loaders and debuggers.

Refer to the *iAPX 286 System Builder User's Guide* for detailed definitions of the iAPX 286 Object Module Format.

Loaders that adhere to Intel's OMFs for loadable modules can load object modules created by Intel's program development tools. There are two basic variations of the iAPX 286 OMF for loadable modules:

- That created by the Binder supports straightforward, single-task modules.
- That created by the Builder supports more complex variations, including multitask modules with shared segments (possibly including shared LDTs), reserved LDT locations, GDT descriptors to be installed as the task is loaded, etc.

Builder not only produces modules intended for use by dynamic loaders but also produces *bootloadable modules* designed for use by bootstrap or initializing loaders. Bootloadable modules use absolute addresses. A boot loader must be able to write to absolute physical locations.

The OMF of each object module consists of a header followed by a sequence of sections of various kinds. The sections relevant to this discussion are those in loadable modules, including

- DESCRP: contains all the descriptors for the module. A loader can use this section as a template for the LDT.
- LODTXT: contains the data and instructions to be loaded. A loader fills allocated memory segments from the records of LODTXT.
- DESNAM: provides symbolic names for segments. A loader can use these symbolic names to provide a load-time interface for making changes to segment parameters (for example, decreasing segment limit of an expand-down stack segment to provide more stack space).

Flow of Loader

A loader must take different action depending on whether the loadable modules are created by the Binder or by the Builder. The main difference in loading the two types of OMF is the source of information for descriptor tables. In the case of Binder OMFs, the DESCRP section is in the format of an LDT. In the case of Builder-created OMFs, information for the GDT, IDT, and LDTs comes from various LODTXT records. A Boolean in the header of a loadable module distinguishes between Builder-created and Binder-created modules.

FLOW FOR BINDER-CREATED MODULE

1. Allocate space for LDT segment. The size of the segment is eight times the value of the descriptor count field of the module header. Put LDT descriptor in GDT.
2. Read the DESCR section into the LDT segment.
3. Allocate space for all segments specified in the LDT, updating the base-address fields.
4. Allocate space for the TSS. (Step 3 does not allocate the TSS because the DESCR section does not contain a TSS descriptor.) Put TSS descriptor in GDT.
5. Read first LODTXT section into TSS segment. (The first LODTXT record is always a TSS.)
6. Put LDT selector into TSS.
7. If the LODTXT section is exhausted, the task is completely loaded. Jump to the TSS.
8. Read next LODTXT record into proper segment. Go to step 7.

FLOW FOR BUILDER-CREATED MODULE

1. Allocate temporary space to hold all entries from the DESCRP section.
2. Read the DESCRP section into this space.
3. Allocate physical segments for all DESCRP entries (except for gates).
4. Read beginning of LODTXT record.
5. Examine descriptor name of LODTXT record. The selector name is either a special identifier for the GDT or IDT, or it is an index into the DESCRP entries. The type field of a DESCRP entry distinguishes LDT segments from other types.
   a. If the LODTXT record defines entries of the GDT, then, for each entry in the record, obtain the descriptor from DESCRP and install it in the GDT.
   b. If the LODTXT record defines entries of the IDT, then, for each entry in the record, obtain the descriptor from DESCRP and install it in the IDT.

c. If the LODTXT record defines an LDT, then, for each entry in the record, obtain the descriptor from DESCRP and install it in the LDT.

d. If the LODTXT record does not apply to a descriptor table, read the LODTXT record directly into the selected segment.

6. If more LODTXT records remain, go to step 4.
7. Free the space used for DESCRP entries.
8. The task or tasks are loaded. Jump to the TSS identified in the module header.

LOAD-TIME BINDING

As with any system, it is possible on the iAPX 286 to delay many binding operations until load time by incorporating binding functions into the loader. The call gates of the iAPX 286 architecture, however, provide an especially convenient and efficient way to delay one specific binding operation: namely, binding of calls to operating-system primitives. Load-time binding to operating-system primitives is an advantage when

- The operating system is under development, and GDT locations of operating-system segment descriptors are subject to change
- Programs may execute under different operating systems that have different GDT layouts

Data structures for implementing load-time binding are

- An export module of specially marked, dummy gates for operating system primitives. Figure 11-4 shows how you can create such a module through Builder specifications. The NOT PRESENT specification marks the gates by resetting the present bit. Gates used for this purpose must reside in the LDT so that Binder (which works only with LDT descriptors) can use them. You need to update this export file only when you change the number or names of operating-system gates.
- A table of actual gates for the operating-system primitives. This table must contain, for each primitive, the gate name, a GDT selector for the segment in which the primitive resides, and the entry point (offset) within the segment. It is most convenient to take such information from a Builder export file that exports the gates for operating-system primitives. The example specifications shown in figure 11-3 create such an export file.

Figure 11-5 shows the data flow for load-time binding. By using the gate name from the DESNAM section, the loader associates a marked gate in the LDT with its gate name. Given the gate name, it looks up the actual binding information.

EXAMPLE LOADER

Figure 11-6 shows an example of a loader that reads the iAPX 286 OMF, resolves references to operating-system primitives, and calls kernel procedures to create a new task. In the interest of simplicity, this example recognizes only the single-task OMF produced by the Binder, and all error checking is omitted.

This loader calls the kernel procedures CREATE_LDT, LOAD_LDT, LOAD_LDT_GATE, CREATE_TASK, RESERVE_SLOTS, and CHANGE_AR, which are not shown. CREATE_LDT allocates an LDT segment for a new task and installs its descriptors in two of the four GDT slots specified by TASK_SEL. LOAD_LDT transfers a descriptor from the LDT of the loader to the LDT of the new task. LOAD_LDT_GATE places a gate in the LDT of the new task. CREATE_TASK

```
system-ID  iAPX286 SYSTEM BUILDER, Vx.y

INPUT FILES:  :F1:NUC.LNK
OUTPUT FILE:  (none)
CONTROLS SPECIFIED:  TITLE(Gates for Load-Time Binding) NOBOOTLOAD NOOBJECT
                     BUILDFILE(:F1:XOSGAT.BLD) PRINT(:F1:XOSGAT.MP2)


BUILD FILE: :F1:XOSGAT.BLD

    1           EXAMPLE_OS;
    2
    3           SEGMENT NUCLEUS_CODE (DPL=0), NUCLEUS_DATA (DPL=0),
    4                   NUCLEUS.STACK(DPL=0);
    5
    6           GATE
    7            XQ_RESERVE_SLOTS    (ENTRY=RESERVE_SLOTS,    DPL=3, NOT PRESENT),
    8            XQ_RELINQUISH_SLOTS (ENTRY=RELINQUISH_SLOTS, DPL=3, NOT PRESENT),
    9            XQ_ALLOCATE         (ENTRY=ALLOCATE,         DPL=3, NOT PRESENT),
   10            XQ_FREE_SEG         (ENTRY=FREE_SEG,         DPL=3, NOT PRESENT),
   11            XQ_CREATE_ALIAS     (ENTRY=CREATE_ALIAS,     DPL=3, NOT PRESENT),
   12            XQ_CHANGE_AR        (ENTRY=CHANGE_AR,        DPL=3, NOT PRESENT),
   13            XQ_WAIT_SEMAPHORE   (ENTRY=WAIT_SEMAPHORE,   DPL=3, NOT PRESENT),
   14            XQ_SIGNAL_SEMAPHORE (ENTRY=SIGNAL_SEMAPHORE, DPL=3, NOT PRESENT),
   15            XQ_SEND_MESSAGE     (ENTRY=SEND_MESSAGE,     DPL=3, NOT PRESENT),
   16            XQ_RECEIVE_MESSAGE  (ENTRY=RECEIVE_MESSAGE,  DPL=3, NOT PRESENT);
   17
   18           TABLE
   19            GDT (ENTRY=(NUCLEUS_CODE, NUCLEUS_DATA, NUCLEUS.STACK)),
   20
   21            DUMMY (ENTRY=(
   22              XQ_RESERVE_SLOTS,
   23              XQ_RELINQUISH_SLOTS,
   24              XQ_ALLOCATE,
   25              XQ_FREE_SEG,
   26              XQ_CREATE_ALIAS,
   27              XQ_CHANGE_AR,
   28              XQ_WAIT_SEMAPHORE,
   29              XQ_SIGNAL_SEMAPHORE,
   30              XQ_SEND_MESSAGE,
   31              XQ_RECEIVE_MESSAGE));
   32
   33            EXPORT #:F1:XOSGAT.LB2 (KERNEL (
   34              XQ_RESERVE_SLOTS,
   35              XQ_RELINQUISH_SLOTS,
   36              XQ_ALLOCATE,
   37              XQ_FREE_SEG,
   38              XQ_CREATE_ALIAS,
   39              XQ_CHANGE_AR,
   40              XQ_WAIT_SEMAPHORE,
   41              XQ_SIGNAL_SEMAPHORE,
   42              XQ_SEND_MESSAGE,
   43              XQ_RECEIVE_MESSAGE));
   44
   45           END
BUILD FILE PROCESSING COMPLETED
```

Figure 11-4. Specifying Dummy Gate Exports

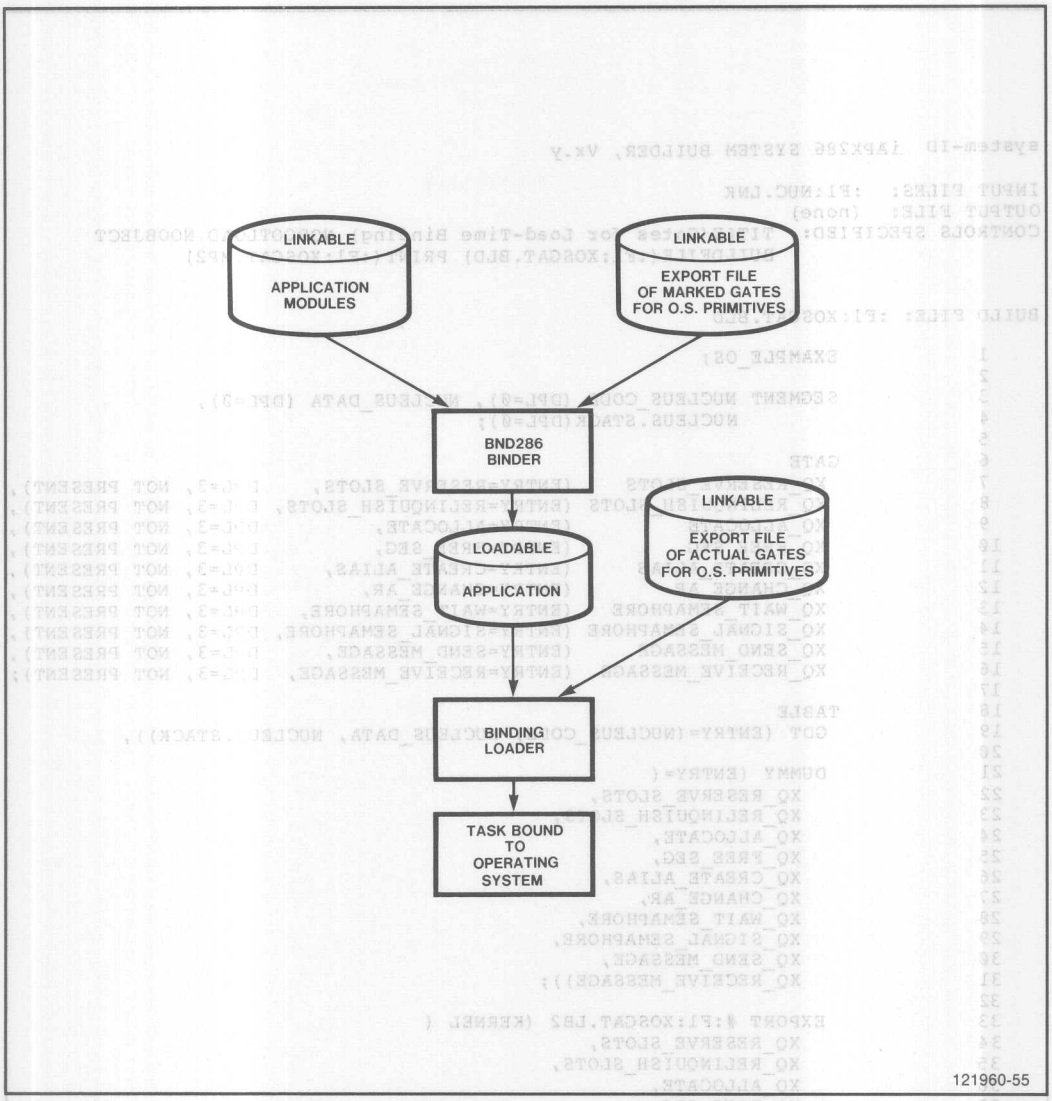


Figure 11-5. Strategy for Load-Time Binding

finishes creation of the new task by creating a TSS from a template in the loader and placing the new task in the scheduler's queues. RESERVE_SLOTS uses techniques such as those illustrated in Chapter 2 to allocate descriptor table slots. CHANGE_AR is used to place the correct type code into the writable data-segment descriptor used by the loader to fill a segment.

The module BOND (shown in figure 11-7) shows the handling of the table of actual gates for operating-system primitives. The procedure GET_LOAD_FILE is not defined here. It simply fetches the file specification for a loadable module from, for example, the console or the command line interpreter.

```
PL/M-286 COMPILER    960-515                    date              PAGE    1


system-ID  PL/M-286 Vx.y COMPILATION OF MODULE LOADER
OBJECT MODULE PLACED IN :F1:LOADER.OBJ
COMPILER INVOKED BY:  PLM286.86 :F1:LOADER.PLM DEBUG


             $ PAGEWIDTH(71) TITLE('960-515')
             $ INCLUDE (:F1:NUCSUB.PLM)
         =   $ NOLIST
             $ INCLUDE (:F1:UDISUB.PLM)
         =   $ NOLIST
             $ COMPACT

  1          LOADER:  DO;

             /*************************************************/
             /*                  Language Extensions          */

  2   1      DECLARE BOOLEAN    LITERALLY 'BYTE',
                     FALSE      LITERALLY '0',
                     TRUE       LITERALLY '0FFH',
                     TOKEN      LITERALLY 'WORD',
                     CONNECTION LITERALLY 'TOKEN',
                     OFFSET     LITERALLY 'WORD',
                     FOREVER    LITERALLY 'WHILE TRUE';

             /*************************************************/
             /*                    Externals                  */

  3   1      RESERVE_SLOTS: PROCEDURE(TABLE,COUNT,SLOT_PTR,EXCEP_PTR)
               EXTERNAL;
  4   2       DECLARE TABLE WORD, COUNT WORD, (SLOT_PTR,EXCEP_PTR)
               POINTER;
  5   2       END RESERVE_SLOTS;

  6   1      CHANGE_AR: PROCEDURE (SLOT,RIGHTS,EXCEP_PTR) EXTERNAL;
  7   2       DECLARE SLOT SELECTOR, RIGHTS BYTE, EXCEP_PTR POINTER;
  8   2       END CHANGE_AR;

  9   1      ALLOCATE: PROCEDURE (SLOT,RIGHTS,SIZE,EXCEP_PTR)
               EXTERNAL;
 10   2       DECLARE SLOT SELECTOR, RIGHTS BYTE, SIZE WORD,
                EXCEP_PTR POINTER;
 11   2       END ALLOCATE;

 12   1      FREE_SEG: PROCEDURE (SLOT, EXCEP_PTR) EXTERNAL;
 13   2       DECLARE SLOT SELECTOR, EXCEP_PTR POINTER;
 14   2       END FREE_SEG;

 15   1      CREATE_LDT: PROCEDURE(TASK_SEL,SIZE,EXCEP_PTR) EXTERNAL;
 16   2       DECLARE TASK_SEL SELECTOR, SIZE WORD, EXCEP_PTR POINTER;
 17   2       END CREATE_LDT;

 18   1      LOAD_LDT: PROCEDURE (TASK_SEL, NEW_SLOT, OLD_SLOT,
               EXCEP_PTR) EXTERNAL;
 19   2       DECLARE (TASK_SEL, NEW_SLOT, OLD_SLOT) SELECTOR,
               EXCEP_PTR POINTER;
```

Figure 11-6. Binding Loader

```
PL/M-286 COMPILER     960-515                    date                PAGE   2


  20    2       END LOAD_LDT;

  21    1     LOAD_LDT_GATE: PROCEDURE (TASK_SEL, NEW_SLOT, RIGHTS,
                  TARGET, COUNT, EXCEP_PTR) EXTERNAL;
  22    2       DECLARE (TASK_SEL, NEW_SLOT) SELECTOR, RIGHTS BYTE,
                  (TARGET, EXCEP_PTR) POINTER, COUNT BYTE;
  23    2       END LOAD_LDT_GATE;

  24    1     CREATE_TASK: PROCEDURE (TASK_SEL, TEMPLATE, PRIORITY,
                  EXCEP_PTR) EXTERNAL;
  25    2       DECLARE TASK_SEL SELECTOR, (TEMPLATE, EXCEP_PTR)
                  POINTER, PRIORITY BYTE;
  26    2       END CREATE_TASK;

  27    1     GET_LOAD_FILE: PROCEDURE (FILE_SPEC_PTR) EXTERNAL;
  28    2       DECLARE FILE_SPEC_PTR POINTER;
  29    2       END GET_LOAD_FILE;

  30    1     BUILD_BOND_TABLE: PROCEDURE (FILESPEC_PTR, EXCEP_PTR)
                  EXTERNAL;
  31    2       DECLARE (FILESPEC_PTR, EXCEP_PTR) POINTER;
  32    2       END BUILD_BOND_TABLE;

  33    1     FIND_BOND: PROCEDURE (SNAME_PTR, ENTRY_PTR, SEL_PTR,
                  EXCEP_PTR) EXTERNAL;
  34    2       DECLARE (SNAME_PTR, ENTRY_PTR, SEL_PTR, EXCEP_PTR)
                  POINTER;
  35    2       END FIND_BOND;

  36    1     INITIALIZE_SYSTEM: PROCEDURE EXTERNAL;
  37    2       END INITIALIZE_SYSTEM;

  38    1     REPORT: PROCEDURE (EXCEP_PTR) EXTERNAL;
  39    2       DECLARE EXCEP_PTR POINTER;
  40    2       END REPORT;

  41    1     DQ$ATTACH:
                  PROCEDURE (PATH$P, EXCEP$P) CONNECTION EXTERNAL;
  42    2       DECLARE (PATH$P, EXCEP$P) POINTER;
  43    2       END DQ$ATTACH;

  44    1     DQ$DETACH:  PROCEDURE (CONN, EXCEP$P) EXTERNAL;
  45    2       DECLARE CONN CONNECTION, EXCEP$P POINTER;
  46    2       END DQ$DETACH;

  47    1     DQ$OPEN:
                  PROCEDURE (CONN, ACCESS, NUM$BUF, EXCEP$P) EXTERNAL;
  48    2       DECLARE CONN     CONNECTION, (ACCESS, NUM$BUF) BYTE,
                        EXCEP$P  POINTER;
  49    2       END DQ$OPEN;

  50    1     DQ$SEEK:  PROCEDURE
                  (CONN, MODE, LOCATION, EXCEP$P) EXTERNAL;
  51    2       DECLARE CONN CONNECTION, MODE BYTE,
                        LOCATION DWORD, EXCEP$P POINTER;
  52    2       END DQ$SEEK;
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-515                    date          PAGE    3


  53   1       DQ$READ:  PROCEDURE
                 (CONN, BUF$P, COUNT, EXCEP$P) WORD EXTERNAL;
  54   2        DECLARE CONN CONNECTION, COUNT WORD,
                    (BUF$P, EXCEP$P) POINTER;
  55   2        END DQ$READ;

  56   1       DQ$CLOSE:  PROCEDURE (CONN, EXCEP$P)  EXTERNAL;
  57   2        DECLARE CONN CONNECTION, EXCEP$P POINTER;
  58   2        END DQ$CLOSE;


                /*****************************************************/
                /*                      Data                         */

  59   1       DECLARE IN_GDT            LITERALLY '0',
                       IN_LDT            LITERALLY '1',
                       DATA_W LITERALLY '11110010B', /* Access rights:
                     present, DPL=3, expand-up, writable, data segment */
                       DATA_WD LITERALLY '11110110B', /* Access rights:
                  present, DPL=3, expand-down, writable, data segment */
                       READ              LITERALLY '1',
                       DEFAULT_PRIORITY LITERALLY '4',
                       ALLOCATED         LITERALLY '80H',
                       OK                LITERALLY '0',
                       EXCEPTION         LITERALLY 'EXCEP<>OK';

  60   1       DECLARE PATH_NAME (47) BYTE,   /* Disk file to load */
                       LOAD_FILE   CONNECTION,
                       ACTUAL      WORD,
                       EXCEP       WORD,
                       TASK_SLOT   SELECTOR, /* First of GDT slots
                                                for new task      */
                       DCS_SEL     SELECTOR, /* Data or code segment
                                                of new task       */
                       BOND_FILESPEC (*) BYTE
                        INITIAL (11,':F1:XOS.LB2');

  61   1       DECLARE FILE_HEADER  BYTE;

  62   1       DECLARE MODULE_HEADER          STRUCTURE (
                        TOTAL_SPACE             DWORD,
                        DESCR_COUNT             WORD,
                        BUILT                   BYTE,
                        DATE             (8)    BYTE,
                        TIME             (8)    BYTE,
                        CREATOR          (41)   BYTE,
                        TSS_SEL                 WORD,
                        DESCRP_LOC              DWORD,  /* 0 */
                        LODTXT_LOC              DWORD,  /* 1 */
                        IGNORE_1                DWORD,  /* 2 */
                        DESNAM_LOC              DWORD,  /* 3 */
                        IGNORE_2                DWORD,  /* 4 */
                        IGNORE_3                DWORD,  /* 5 */
                        IGNORE_4                DWORD,  /* 6 */
                        LAST_LOC                DWORD,  /* 7 */
                        RESERVED1               DWORD,
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-515                 date          PAGE    4


                      RESERVED2         DWORD);

                /* Table of segment names from DESNAM section of OMF */
  63    1       DECLARE DESNAM_SEL SELECTOR,
                        DESNAM_WIDTH LITERALLY '41',
                        DESNAM BASED DESNAM_SEL (2) STRUCTURE (
                         NAME (DESNAM_WIDTH) BYTE          ),
                        DIX         WORD;    /* Index */

                /* RAM copy of DESCRP section of OMF */
  64    1       DECLARE DESCRP_SEL SELECTOR,
                        SEGDT BASED DESCRP_SEL (2) STRUCTURE (
                         LIMIT       WORD,
                         LO_BASE     WORD,
                         HI_BASE     BYTE, /* Data/code segment descr */
                         RIGHTS      BYTE,
                         RESERVED    WORD              ),
                        GATET BASED DESCRP_SEL (2) STRUCTURE (
                         ENTRY_POINT OFFSET,
                         SEL         SELECTOR,
                         WORD_COUNT  BYTE,     /* Gate descriptor */
                         RIGHTS      BYTE,
                         RESERVED    WORD              ),
                        LIX         WORD;          /* Index */

                /* Template for Task State Segment for new task */
  65    1       DECLARE TSS STRUCTURE (
                         BACKL SELECTOR,
                         SP0 OFFSET, SS0 SELECTOR,
                         SP1 OFFSET, SS1 SELECTOR,
                         SP2 OFFSET, SS2 SELECTOR,
                         IP  OFFSET,
                         FLAG  WORD,
                         (AX, CX, DX, BX, SP, BP, SI, DI,
                          ES, CS, SS, DS, LDT_LINK) SELECTOR   );

                /*******************************************************/
                /*                    Subroutines                      */

  66    1       SECTION_SIZE: PROCEDURE (TOCIX) DWORD;

  67    2        DECLARE TOCIX   WORD; /* Index into Table of Contents */
  68    2        DECLARE TOC(8) DWORD AT (@MODULE_HEADER.DESCRP_LOC),
                        IX      WORD;

                 /* Find the size of an OMF section */
  69    2        DO IX=TOCIX+1 TO 7;
  70    3         IF TOC(IX)<>0 /* Skip by null sections */
  71    3          THEN RETURN TOC(IX)-TOC(TOCIX);
  72    3        END;

  73    2       END SECTION_SIZE;

                /*******************************************************/
  74    1       BUILD_DESNAM_TABLE: PROCEDURE (DESNAM_SIZE);

  75    2        DECLARE DESNAM_SIZE  DWORD;
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-515                    date                PAGE   5


  76   2        DECLARE DESNAM_USED  DWORD,
                       DESNAM_HEADER STRUCTURE
                        (DESCR_IN     WORD,
                         NAME_LENGTH  BYTE);

  77   2        DESNAM_USED=0;
                /* Allocate a segment for the table */
  78   2        CALL ALLOCATE (DESNAM_SEL, DATA_W,
                   DESNAM_WIDTH * (MODULE_HEADER.DESCR_COUNT + 1), @EXCE
                      -P);
  79   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);
                /* Initialize the table */
  81   2        DO DIX=0 TO MODULE_HEADER.DESCR_COUNT;
  82   3         CALL MOVB(@(' '),@DESNAM(DIX).NAME(0),1);
  83   3         CALL MOVB(@DESNAM(DIX).NAME(0),
                        @DESNAM(DIX).NAME(1),DESNAM_WIDTH-1);
  84   3        END;
                /* Read each descriptor name */
  85   2        DO WHILE DESNAM_USED < DESNAM_SIZE;
                  /* Read fixed portion of DESNAM record */
  86   3         ACTUAL=DQ$READ (LOAD_FILE, @DESNAM_HEADER, 3, @EXCEP);
  87   3         IF EXCEPTION THEN CALL REPORT (@EXCEP);
  89   3         DESNAM_USED=DESNAM_USED+ACTUAL;
  90   3         DIX=DESNAM_HEADER.DESCR_IN-1;
  91   3         DESNAM(DIX).NAME(0)=DESNAM_HEADER.NAME_LENGTH;
                   /* Read rest of name into table entry */
  92   3         ACTUAL=DQ$READ(LOAD_FILE,@DESNAM(DIX).NAME(1),
                                 DESNAM(DIX).NAME(0), @EXCEP);
  93   3         IF EXCEPTION THEN CALL REPORT (@EXCEP);
  95   3         DESNAM_USED=DESNAM_USED+ACTUAL;
  96   3        END /* DO LOOP */;

  97   2      END BUILD_DESNAM_TABLE;

                /*************************************************/
  98   1      LOAD_SEGMENTS: PROCEDURE(LODTXT_SIZE);

  99   2        DECLARE LODTXT_SIZE   DWORD;

 100   2        DECLARE LODTXT_HEADER STRUCTURE (
                         LOAD_OFFSET   WORD,
                         DESCR_IN      WORD,
                         LENGTH        WORD      ),
                       COUNT           WORD,
                       LODTXT_USED     DWORD;

 101   2        DECLARE NEW_LDT_SEL   SELECTOR,
                       NEW_LDT_SEL_W WORD AT (@NEW_LDT_SEL),
                       SEG_RIGHTS    BYTE;

 102   2        DECLARE TSS_IN LITERALLY '0FFFDH';

 103   2        LODTXT_USED=0;
                 /* Step thru all LODTXT records */
 104   2        DO WHILE LODTXT_USED<LODTXT_SIZE;
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER    960-515                 date          PAGE   6


                 /* Read in the LODTXT header */
105    3         ACTUAL=DQ$READ(LOAD_FILE,@LODTXT_HEADER,6,@EXCEP);
106    3         IF EXCEPTION THEN CALL REPORT (@EXCEP);
108    3         LODTXT_USED=LODTXT_USED+ACTUAL;
109    3         IF LODTXT_HEADER.DESCR_IN=TSS_IN

110    3          THEN DO; /* Load the Task State Segment */
111    4           ACTUAL=DQ$READ(LOAD_FILE,@TSS,LODTXT_HEADER.LENGTH,
                                  @EXCEP);
112    4           IF EXCEPTION THEN CALL REPORT (@EXCEP);
114    4           LODTXT_USED=LODTXT_USED+ACTUAL;
115    4           END /* loading Task State Segment */;

116    3          ELSE DO; /* Load a data or code segment */
117    4           LIX=LODTXT_HEADER.DESCR_IN-1;
118    4           IF (SEGDT(LIX).RIGHTS AND 06H) = 06H
                     /* expand-down data segment? */
119    4            THEN SEG_RIGHTS=DATA_WD;
120    4            ELSE SEG_RIGHTS=DATA_W;
                     /* Allocate a segment */
121    4           CALL ALLOCATE (DCS_SEL, SEG_RIGHTS,
                      SEGDT(LIX).LIMIT+1, @EXCEP);
122    4           IF EXCEPTION THEN CALL REPORT (@EXCEP);
                     /* Read LODTXT record into segment */
124    4           ACTUAL=DQ$READ(LOAD_FILE,
                        BUILD$PTR(DCS_SEL,LODTXT_HEADER.LOAD_OFFSET),
                                  LODTXT_HEADER.LENGTH, @EXCEP);
125    4           IF EXCEPTION THEN CALL REPORT (@EXCEP);
127    4           LODTXT_USED=LODTXT_USED+ACTUAL;
                     /* Put actual access rights in descriptor */
128    4           CALL CHANGE_AR (DCS_SEL, SEGDT(LIX).RIGHTS, @EXCEP);
129    4           IF EXCEPTION THEN CALL REPORT (@EXCEP);
                     /* Construct selector for slot in new LDT */
                     /* DPL = 3; TI = 1 */
131    4           NEW_LDT_SEL_W = (SHL(LIX,3) OR 07H);
                     /* Transfer descriptor to new LDT */
132    4           CALL LOAD_LDT(TASK_SLOT,NEW_LDT_SEL,DCS_SEL,@EXCEP);
133    4           IF EXCEPTION THEN CALL REPORT (@EXCEP);
                     /* Mark descriptor as allocated */
135    4           SEGDT(LIX).RIGHTS=ALLOCATED;
136    4          END /* loading a data or code segment */;
137    3        END /* stepping thru all LODTXT records */;

138    2      END LOAD_SEGMENTS;

               /*********************************************************/
139    1      LOAD_DESCRP: PROCEDURE;

                /* Allocate a segment for the DESCRP section */
140    2        CALL ALLOCATE (DESCRP_SEL, DATA_W,
                          8*MODULE_HEADER.DESCR_COUNT, @EXCEP);
141    2        IF EXCEPTION THEN CALL REPORT (@EXCEP);
                  /* Step thru all descriptors */
143    2        DO LIX = 0 TO MODULE_HEADER.DESCR_COUNT-1;
                    /* Read LDT entry */
144    3         ACTUAL=DQ$READ (LOAD_FILE, @SEGDT(LIX),
                                  8, @EXCEP);
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-515                 date                PAGE   7


145   3         IF EXCEPTION THEN CALL REPORT (@EXCEP);
                 /* Is it marked with Present bit = 0 ? */
147   3         IF (SEGDT(LIX).RIGHTS AND 80H)=0 THEN
                   /* Is it a call gate? */
148   3          IF (SEGDT(LIX).RIGHTS AND 0FH)=4 /* Type field */
149   3           THEN DO; /* Insert pointer from BOND table */
150   4            CALL FIND_BOND (@DESNAM(LIX).NAME,
                     @GATET(LIX).ENTRY_POINT, @GATET(LIX).SEL,
                     @EXCEP);
151   4            IF EXCEP=OK THEN /* Set present bit. */
152   4              GATET(LIX).RIGHTS=GATET(LIX).RIGHTS OR 80H;
153   4           END /* inserting pointer */;
154   3         END /* stepping through all descriptors */;

155   2       END LOAD_DESCRP;

                /*************************************************/
                /*     Transfer remaining descriptors to new LDT     */

156   1       TRANSFER_REMAINDERS: PROCEDURE;
                 /* Handles descriptors for which there was no
                    LODTXT record (e.g. stacks and gates).      */

157   2          DECLARE NEW_LDT_SEL   SELECTOR,
                         NEW_LDT_SEL_W WORD AT (@NEW_LDT_SEL),
                         SEG_RIGHTS    BYTE;

                   /* Step thru DESCRP entries, skipping LDT alias */
158   2          DO LIX = 2 TO MODULE_HEADER.DESCR_COUNT-1;

159   3           IF (SEGDT(LIX).RIGHTS AND 0FH) = 4
160   3            THEN /* call-gate */ DO;
                      /* Construct selector for slot in new LDT */
161   4             NEW_LDT_SEL_W = (SHL(LIX,3) OR 07H);
                      /* Transfer descriptor to new LDT */
162   4             CALL LOAD_LDT_GATE (TASK_SLOT, NEW_LDT_SEL,
                        GATET(LIX).RIGHTS,
                        BUILD$PTR(GATET(LIX).SEL,
                                  GATET(LIX).ENTRY_POINT),
                        GATET(LIX).WORD_COUNT, @EXCEP);
163   4             IF EXCEPTION THEN CALL REPORT (@EXCEP);
165   4            END /* call gate */;

166   3           ELSE IF (SEGDT(LIX).RIGHTS AND 10H) <> 0
167   3            THEN /* unallocated data or code segment */ DO;
168   4             IF (SEGDT(LIX).RIGHTS AND 06H) = 06H
                      /* expand-down data segment? */
169   4              THEN SEG_RIGHTS=DATA_WD;
170   4              ELSE SEG_RIGHTS=DATA_W;
                      /* Allocate a segment */
171   4             CALL ALLOCATE (DCS_SEL, SEG_RIGHTS,
                        SEGDT(LIX).LIMIT+1, @EXCEP);
172   4             IF EXCEPTION THEN CALL REPORT (@EXCEP);
                      /* Put actual access rights in descriptor */
174   4             CALL CHANGE_AR (DCS_SEL, SEGDT(LIX).RIGHTS, @EXCEP);
175   4             IF EXCEPTION THEN CALL REPORT (@EXCEP);
                      /* Construct selector for slot in new LDT */
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER    960-515                 date          PAGE   8


                    /* DPL = 3; TI = 1 */
177   4              NEW_LDT_SEL_W = (SHL(LIX,3) OR 07H);
                     /* Transfer descriptor to new LDT */
178   4             CALL LOAD_LDT(TASK_SLOT,NEW_LDT_SEL,DCS_SEL,@EXCEP);
179   4             IF EXCEPTION THEN CALL REPORT (@EXCEP);
181   4            END /* unallocated data or code segment */;

182   3           END /* stepping thru DESCRP entries */;

183   2         END TRANSFER_REMAINDERS;

              /*******************************************************/
              /*                      Main Line                      */

184   1           CALL INITIALIZE_SYSTEM;

185   1           CALL BUILD_BOND_TABLE (@BOND_FILESPEC,@EXCEP);
186   1           IF EXCEPTION THEN CALL REPORT (@EXCEP);
188   1           CALL RESERVE_SLOTS (IN_LDT, 1, @DCS_SEL, @EXCEP);
189   1           IF EXCEPTION THEN CALL REPORT (@EXCEP);
191   1           CALL RESERVE_SLOTS (IN_LDT, 1, @DESCRP_SEL, @EXCEP);
192   1           IF EXCEPTION THEN CALL REPORT (@EXCEP);
194   1           CALL RESERVE_SLOTS (IN_LDT, 1, @DESNAM_SEL, @EXCEP);
195   1           IF EXCEPTION THEN CALL REPORT (@EXCEP);

197   1         DO FOREVER;

198   2         GET_NAME:
                CALL GET_LOAD_FILE (@PATH_NAME); /* May wait */
199   2           LOAD_FILE=DQ$ATTACH (@PATH_NAME, @EXCEP);
200   2           IF EXCEPTION THEN GOTO GET_NAME;
202   2           CALL DQ$OPEN (LOAD_FILE, READ, 1, @EXCEP);
203   2           IF EXCEPTION THEN CALL REPORT (@EXCEP);
                  /* Read file header. */
205   2           ACTUAL=DQ$READ (LOAD_FILE, @FILE_HEADER, 1, @EXCEP);
206   2           IF EXCEPTION THEN CALL REPORT (@EXCEP);
                  /* Read loadable-module header */
208   2           ACTUAL=DQ$READ (LOAD_FILE, @MODULE_HEADER,
                                  SIZE(MODULE_HEADER), @EXCEP);
209   2           IF EXCEPTION THEN CALL REPORT (@EXCEP);

                  /* Process DESNAM section of OMF */
211   2           CALL DQ$SEEK (LOAD_FILE,2,MODULE_HEADER.DESNAM_LOC,
                                @EXCEP);
212   2           IF EXCEPTION THEN CALL REPORT (@EXCEP);
214   2           CALL BUILD_DESNAM_TABLE (SECTION_SIZE(3));

                  /* Tell OS to allocate an LDT */
215   2           CALL RESERVE_SLOTS (IN_GDT, 4, @TASK_SLOT, @EXCEP);
216   2           IF EXCEPTION THEN CALL REPORT (@EXCEP);
218   2           CALL CREATE_LDT (TASK_SLOT, MODULE_HEADER.DESCR_COUNT,
                                  @EXCEP);
219   2           IF EXCEPTION THEN CALL REPORT (@EXCEP);

                  /* Process DESCRP section */
221   2           CALL DQ$SEEK (LOAD_FILE,2,MODULE_HEADER.DESCRP_LOC,
                                @EXCEP);
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-515                 date     PAGE   9


 222   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);
 224   2        CALL LOAD_DESCRP;

                  /* Process LODTXT section */
 225   2        CALL DQ$SEEK (LOAD_FILE,2,MODULE_HEADER.LODTXT_LOC,
                           @EXCEP);
 226   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);
 228   2        CALL LOAD_SEGMENTS(SECTION_SIZE(1));

 229   2        CALL TRANSFER_REMAINDERS;

                  /* Tell OS to create the new task */
 230   2        CALL CREATE_TASK(TASK_SLOT, @TSS, DEFAULT_PRIORITY,
                                 @EXCEP);
 231   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);

 233   2        CALL DQ$CLOSE   (LOAD_FILE, @EXCEP);
 234   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);
 236   2        CALL DQ$DETACH (LOAD_FILE, @EXCEP);
 237   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);

 239   2        CALL FREE_SEG (DESCRP_SEL, @EXCEP);
 240   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);
 242   2        CALL FREE_SEG (DESNAM_SEL, @EXCEP);
 243   2        IF EXCEPTION THEN CALL REPORT (@EXCEP);

 245   2      END /* FOREVER */;

                /*****************************************************/
 246   1      END LOADER;


MODULE INFORMATION:

     CODE AREA SIZE     = 08B2H   2226D
     CONSTANT AREA SIZE = 0001H      1D
     VARIABLE AREA SIZE = 00FFH    255D
     MAXIMUM STACK SIZE = 0018H     24D
     508 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     11KB MEMORY USED      (11%)
     0KB DISK SPACE USED

END OF PL/M-286 COMPILATION
```

Figure 11-6. Binding Loader (Cont'd.)

```
PL/M-286 COMPILER    960-521                    date          PAGE   1


system-ID  PL/M-286 Vx.y  COMPILATION OF MODULE BOND
OBJECT MODULE PLACED IN :F1:BOND.OBJ
COMPILER INVOKED BY:  PLM286.86 :F1:BOND.PLM DEBUG

                $ PAGEWIDTH(71) TITLE('960-521')
                $ INCLUDE (:F1:NUCSUB.PLM)
          =     $ NOLIST
                $ INCLUDE (:F1:UDISUB.PLM)
          =     $ NOLIST
                $ COMPACT
   1            BOND:  DO;
                /*************************************************/
                /*                Language Extensions            */
   2    1       DECLARE BOOLEAN   LITERALLY 'BYTE',
                        FALSE     LITERALLY '0',
                        TRUE      LITERALLY '0FFH',
                        TOKEN     LITERALLY 'WORD',
                        CONNECTION LITERALLY 'TOKEN',
                        OFFSET    LITERALLY 'WORD';
                /*************************************************/
                /*                   Externals                   */
   3    1       RESERVE_SLOTS: PROCEDURE(TABLE,COUNT,SLOT_PTR,EXCEP_PTR)
                  EXTERNAL;
   4    2        DECLARE TABLE WORD, COUNT WORD, (SLOT_PTR, EXCEP_PTR)
                  POINTER;
   5    2        END RESERVE_SLOTS;

   6    1       GET_GATE_POINTER: PROCEDURE (GATE_SEL, SEG_SEL_PTR,
                  SEG_OFFSET_PTR, EXCEP_PTR) EXTERNAL;
                  /* Returns selector and offset from a gate descriptor */
   7    2        DECLARE GATE_SEL SELECTOR,
                   (SEG_SEL_PTR, SEG_OFFSET_PTR, EXCEP_PTR) POINTER;
   8    2        END GET_GATE_POINTER;

   9    1       ALLOCATE: PROCEDURE (SLOT,RIGHTS,SIZE,EXCEP_PTR)
                  EXTERNAL;
  10    2        DECLARE SLOT SELECTOR, RIGHTS BYTE, SIZE WORD,
                   EXCEP_PTR POINTER;
  11    2        END ALLOCATE;

  12    1       REPORT: PROCEDURE (EXCEP_PTR) EXTERNAL;
  13    2        DECLARE EXCEP_PTR POINTER;
  14    2        END REPORT;

  15    1       DQ$ATTACH:
                  PROCEDURE (PATH$P, EXCEP$P) CONNECTION EXTERNAL;
  16    2        DECLARE (PATH$P, EXCEP$P) POINTER;
  17    2        END DQ$ATTACH;

  18    1       DQ$DETACH:  PROCEDURE (CONN, EXCEP$P) EXTERNAL;
```

Figure 11-7. BOND Module of Binding Loader

```
PL/M-286 COMPILER     960-521                 date          PAGE  2


 19   2        DECLARE CONN CONNECTION, EXCEP$P POINTER;
 20   2        END DQ$DETACH;

 21   1      DQ$OPEN:
               PROCEDURE (CONN, ACCESS, NUM$BUF, EXCEP$P) EXTERNAL;
 22   2        DECLARE CONN     CONNECTION, (ACCESS, NUM$BUF) BYTE,
                       EXCEP$P  POINTER;
 23   2        END DQ$OPEN;

 24   1      DQ$SEEK:  PROCEDURE
                (CONN, MODE, LOCATION, EXCEP$P) EXTERNAL;
 25   2        DECLARE CONN CONNECTION, MODE BYTE,
                       LOCATION DWORD, EXCEP$P POINTER;
 26   2        END DQ$SEEK;

 27   1      DQ$READ:  PROCEDURE
                (CONN, BUF$P, COUNT, EXCEP$P) WORD EXTERNAL;
 28   2        DECLARE CONN CONNECTION, COUNT WORD,
                       (BUF$P, EXCEP$P) POINTER;
 29   2        END DQ$READ;

 30   1      DQ$CLOSE:  PROCEDURE (CONN, EXCEP$P)  EXTERNAL;
 31   2        DECLARE CONN CONNECTION, EXCEP$P POINTER;
 32   2        END DQ$CLOSE;

              /*******************************************************/
              /*                      Data                           */

 33   1      DECLARE IN_LDT  LITERALLY '1',
                      DATA_W  LITERALLY '11110010B', /* Access rights:
                   present, DPL=3, expand-up, writable, data segment */
                      READ    LITERALLY '1',
                      EQUALS  LITERALLY '0FFFFH',
                      OK      LITERALLY '0',
                      EXCEPTION LITERALLY 'EXCEP<>OK';

 34   1      DECLARE BOND_FILE   CONNECTION,
                      ACTUAL     WORD,
                      SEL        SELECTOR, /* for type conversion */
                      WSEL       WORD AT (@SEL);

 35   1      DECLARE FILE_HEADER  BYTE;

 36   1      DECLARE MODULE_HEADER    STRUCTURE (
                      TOTAL_LENGTH     DWORD,
                      SEGMENT_COUNT    WORD,
                      GATE_COUNT       WORD,
                      PUB_COUNT        WORD,
                      EXT_COUNT        WORD,
                      LINKED           BYTE,
                      DATE        (8)  BYTE,
                      TIME        (8)  BYTE,
                      MODULE_NAME (41) BYTE,
                      CREATOR     (41) BYTE,
                      IGNORE1     (6)  DWORD,
                      PUBDEF_LOC       DWORD,
                      PUBDEF_LENGTH    DWORD,
```

Figure 11-7. BOND Module of Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-521                 date          PAGE  3


                    IGNORE2         (8)     DWORD);
                /* Table of actual locations of O.S. primitives */
  37   1        DECLARE BOND_SEL SELECTOR,
                       BOND_BASED BOND_SEL (2) STRUCTURE (
                        GATE_NAME(41) BYTE,
                        ENTRY_POINT   OFFSET,
                        GDT_SEL       SELECTOR ),
                       BIX           WORD;          /* Index */

  38   1        DECLARE PUBDEF STRUCTURE
                       (ENTRY_POINT    WORD,
                        GDT_IN         WORD,
                        IGNORE         WORD,
                        WORD_COUNT     BYTE,
                        LENGTH         BYTE);
                /*************************************************/
                /*              Subroutines                      */

                 /* Create table of actual GDT selectors and
                    entry points for O.S. primitives. */
  39   1        BUILD_BOND_TABLE: PROCEDURE (BOND_NAME_PTR, EXCEP_PTR)
                   PUBLIC;
  40   2         DECLARE (BOND_NAME_PTR, EXCEP_PTR) POINTER;
  41   2         DECLARE EXCEP BASED EXCEP_PTR WORD;

                    /* Initialize file */
  42   2         BOND_FILE=DQ$ATTACH (BOND_NAME_PTR,@EXCEP);
  43   2         IF EXCEPTION THEN RETURN;
  45   2         CALL DQ$OPEN (BOND_FILE, READ, 1, @EXCEP);
  46   2         IF EXCEPTION THEN CALL REPORT (@EXCEP);
                    /* Read file header */
  48   2         ACTUAL=DQ$READ (BOND_FILE,@FILE_HEADER,1,@EXCEP);
  49   2         IF EXCEPTION THEN CALL REPORT (@EXCEP);
                    /* Read module header */
  51   2         ACTUAL=DQ$READ (BOND_FILE,@MODULE_HEADER,
                                 SIZE(MODULE_HEADER), @EXCEP);
  52   2         IF EXCEPTION THEN CALL REPORT (@EXCEP);
                    /* Get space for table */
  54   2         CALL RESERVE_SLOTS (IN_LDT,1,@BOND_SEL,@EXCEP);
  55   2         IF EXCEPTION THEN CALL REPORT (@EXCEP);
  57   2         CALL ALLOCATE (BOND_SEL, DATA_W,
                   (MODULE_HEADER.PUB_COUNT+1)*SIZE(BOND(0)),@EXCEP);
  58   2         IF EXCEPTION THEN CALL REPORT (@EXCEP);

                    /* Read the PUBDEF section */
                    /* (Locations are relative to beginning of module,
                        not beginning of file.  Assume module at 1.)  */
  60   2         CALL DQ$SEEK (BOND_FILE,2,MODULE_HEADER.PUBDEF_LOC+1,
                                @EXCEP);
  61   2         IF EXCEPTION THEN CALL REPORT (@EXCEP);
                    /* Loop thru the PUBDEF entries */
  63   2         DO BIX = 0 TO MODULE_HEADER.PUB_COUNT-1;
                      /* Read fixed part of PUBDEF record */
  64   3          ACTUAL=DQ$READ(BOND_FILE,@PUBDEF, 8, @EXCEP);
```

Figure 11-7. BOND Module of Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-521                    date                PAGE   4


  65   3          IF EXCEPTION THEN CALL REPORT (@EXCEP);
                   /* Convert index into GDT selector */
  67   3          WSEL=SHL(PUBDEF.GDT_IN,3);
                   /* Extract gate selector and offset from GDT */
  68   3          CALL GET_GATE_POINTER (SEL, @BOND(BIX).GDT_SEL,
                       @BOND(BIX).ENTRY_POINT, @EXCEP);
  69   3          IF EXCEPTION THEN CALL REPORT (@EXCEP);
                   /* Read variable-length name */
  71   3          BOND(BIX).GATE_NAME(0)=PUBDEF.LENGTH;
  72   3          ACTUAL=DQ$READ(BOND_FILE,@BOND(BIX).GATE_NAME(1),
                                        PUBDEF.LENGTH,@EXCEP);
  73   3          IF EXCEPTION THEN CALL REPORT (@EXCEP);

  75   3         END /* looping thru PUBDEF entries */;

  76   2        END BUILD_BOND_TABLE;

                /*****************************************************/

  77   1        FIND_BOND: PROCEDURE (SNAME_PTR, ENTRY_PTR, SEL_PTR,
                 EXCEP_PTR) PUBLIC;

                 /* Search BOND table for given name.
                    Return items from found entry.          */

  78   2         DECLARE (SNAME_PTR, ENTRY_PTR, SEL_PTR, EXCEP_PTR)
                  POINTER;
  79   2         DECLARE SNAME       BASED SNAME_PTR (41) BYTE,
                         ENTRY_POINT BASED ENTRY_PTR WORD,
                         GDT_SEL     BASED SEL_PTR   SELECTOR,
                         EXCEP       BASED EXCEP_PTR WORD;

  80   2         DO BIX = 0 TO MODULE_HEADER.PUB_COUNT-1;
  81   3          IF SNAME(0)=BOND(BIX).GATE_NAME(0) /* Same length? */
                   THEN /* Compare characters */
  82   3            IF EQUALS=CMPB(@SNAME(1),@BOND(BIX).GATE_NAME(1),
                                  SNAME(0))
  83   3             THEN DO;
  84   4              ENTRY_POINT=BOND(BIX).ENTRY_POINT;
  85   4              GDT_SEL=BOND(BIX).GDT_SEL;
  86   4              EXCEP=OK;
  87   4              RETURN;
  88   4             END;
  89   3         END;
  90   2         EXCEP=NOT OK;
  91   2         RETURN;

  92   2        END FIND_BOND;

                /*****************************************************/

  93   1        END BOND;


MODULE INFORMATION:
```

Figure 11-7. BOND Module of Binding Loader (Cont'd.)

```
PL/M-286 COMPILER     960-521                    date                PAGE   5


     CODE AREA SIZE      = 02CBH     715D
     CONSTANT AREA SIZE  = 0000H       0D
     VARIABLE AREA SIZE  = 00C2H     194D
     MAXIMUM STACK SIZE  = 001CH      28D
     245 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     96KB MEMORY AVAILABLE
     8KB MEMORY USED   (8%)
     0KB DISK SPACE USED

END OF PL/M-286 COMPILATION
```

Figure 11-7. BOND Module of Binding Loader (Cont'd.)

Numerics Processor Extension 12

12 Numerics Processor Extension

CHAPTER 12
NUMERICS PROCESSOR EXTENSION

The iAPX 286/20 is a configuration of chips consisting of an 80286 CPU and an 80287 Numerics Processor Extension (NPX). With these two cooperating processors it is possible to construct powerful numerics processing systems, but the operating system must multiplex the 80287 among the tasks that use it. If the system does not include an 80287, you may choose to have the operating system emulate its functions.

iAPX 286/20 NUMERICS PROCESSING FEATURES

Several features of the iAPX 286/20 are of special interest to operating-system designers and programmers. You can find more details on how to use the iAPX 286/20 in the *iAPX 286 Programmer's Reference Manual.*

ESCAPE Instructions

The 80287 NPX extends the instruction set of the iAPX 286 by over fifty opcodes. The CPU identifies the extended instruction set by the bit pattern 11011B in the high-order five bits of the first byte of the instruction. Instructions thus marked are called ESCAPE or ESC instructions.

The CPU performs some functions upon encountering an ESC instruction, before sending the instruction to the NPX. Those functions that are of interest to the operating system include

- Testing the emulation mode (EM) flag to determine whether NPX functions are being emulated by software.
- Testing the TS flag to determine whether there has been a context change since the last ESC instruction.
- For some ESC instructions, testing the $\overline{\text{ERROR}}$ pin to determine whether an error condition exists at the NPX as a result of a previous ESC instruction.

The *ASM286 Assembly Language Reference Manual* provides more information on each 80287 instruction.

Emulation Mode Flag (EM)

The EM bit of the 80286 machine status word (MSW) indicates to the CPU whether NPX functions are to be emulated. If the processor finds EM set when executing an ESC instruction, it causes trap 7, giving the exception handler an opportunity to emulate the functions of an 80287. The EM flag can be changed with the aid of LMSW (load machine status word) instruction (legal only at privilege level 0 (PL 0)) and tested with the aid of the SMSW (store machine status word). The built-in variable MACHINE$STATUS gives PL/M-286 programs access to the MSW.

The EM bit also controls the function of the WAIT instruction. If the processor finds EM set while executing a WAIT, the processor does not check the $\overline{\text{ERROR}}$ pin for an error indication.

Note that EM must never be set concurrently with MP.

The MP bit of the 80286 machine status word (MSW) indicates to the CPU whether an 80287 NPX is actually attached. The MP flag controls the function of the WAIT instruction. If, when executing a WAIT instruction, the CPU finds MP set, then it tests TS; it does not otherwise test TS during a WAIT instruction. If it finds TS set under these conditions, the CPU causes trap 7.

Note that MP must never be set concurrently with EM.

Task Switched Flag (TS)

The TS bit of the MSW helps to determine when the context of the 80287 NPX does not match that of the 80286 CPU. The CPU sets TS each time it performs a task switch (whether triggered by software or by hardware interrupt). If, when interpreting one of the ESC instructions, the CPU finds TS already set, it causes trap 7. The MP flag also relates to TS.

The CLTS instruction (legal only at PL 0) resets TS.

WAIT Instruction

The WAIT instruction is not an ESC instruction, but WAIT causes the CPU to perform some of the same tests that it performs upon encountering an ESC instruction:

- The CPU waits until the NPX no longer asserts the $\overline{\text{BUSY}}$ pin. You can therefore use WAIT to synchronize the CPU with the NPX.
- The CPU tests the $\overline{\text{ERROR}}$ pin (if EM is not set). You can therefore use WAIT to cause trap 16 if an error is pending from a previous ESC instruction. (The CPU makes this test only after $\overline{\text{BUSY}}$ goes inactive.) Note that, if no 80287 is present, the $\overline{\text{ERROR}}$ pin should be tied inactive to prevent WAIT from causing spurious traps.

Summary

Table 12-1 summarizes functions of the ESC and WAIT instructions that depend on setting of the MP and EM flags.

INITIALIZATION

During its initialization phase the operating system must

- Set flags in the MSW to reflect the numerics processing environment
- Reset the 80287 (if present)
- Switch the 80287 into protected mode

You can use a configuration parameter to communicate the numerics processing environment to the operating system. The FNINIT instruction (INIT$REAL$MATH$UNIT in PL/M-286) resets the 80287, and the FSETPM instruction places the 80287 into protected mode.

Table 12-1. Interpretation of MP and EM Flags

Instruction	EM Flag	Set	Reset
	MP Flag	Reset	Set
WAIT	TRAP 7?	N	if TS=1
	TEST $\overline{\text{BUSY}}$?	Y	Y
	TEST $\overline{\text{ERROR}}$? (TRAP 16)	Y	Y
ESCAPE	TRAP 7?	Y	if TS=1
	TEST $\overline{\text{BUSY}}$?	N	Y
	TEST $\overline{\text{ERROR}}$? (TRAP 16)	N	Y

TASK STATE

When a task uses the 80287 NPX, the operating system has two additional concerns in keeping track of the task's state:

- The task database must be expanded to include 47 words of state information for the 80287.
- The operating system must change 80287 state when a different task attempts to use the 80287.

The state of the 80287 consists of 47 words. Saving and restoring the 80287 state is therefore a relatively expensive operation that should not be performed any more frequently than necessary. Typically, only a few of the active tasks in a system use the NPX; therefore, it would be wasteful to save and update the state information with every task switch. It is preferable for the operating system to record which task is using the 80287 and to swap state information only when some other task attempts to use the 80287.

The 80286 supports sharing of the 80287 by providing the TS flag. The processor automatically sets the TS every time a task switch occurs. The first use of an ESC or WAIT instruction when the TS is set causes trap 7. This enables the operating system to keep track of the task to which the NPX is assigned at any given time and to change 80287 state when necessary.

NUMERICS EXCEPTIONS

Three interrupt vector positions are reserved for exceptions that relate to numerics processing. Interrupts for these vectors are not maskable.

Interrupt 7—Processor Extension Not Available (NM)

This exception occurs in either of two conditions:

- The CPU encounters an ESC instruction and EM is set. In this case, the exception handler should emulate the instruction that caused the exception. TS may also be set.

- The CPU encounters either the WAIT instruction or an ESC instruction when both MP and TS are set. In this case, the exception handler should update the state of the NPX if necessary.

EMULATION

The return link points to the first byte of the ESC instruction (or to the prefix byte, if any). As the emulator decodes the ESC instruction, it should step the return pointer so that, at the end of the emulation routine, the return from the exception handler causes execution to resume at the first instruction following the ESC instruction.

UPDATING STATE

To make sure that the state of the NPX corresponds to the current task, the operating system should implement the concept of "ownership" of the NPX. Ownership can be indicated by a Boolean in the task database (TDB). The operating system must ensure that only one task at a time is marked as the owner of the NPX. The exception handler should follow these steps:

1. Use the CLTS instruction to reset TS.
2. Return if the current task owns the NPX.
3. Use the FSAVE ESC instruction (PL/M-286 SAVE$REAL$STATUS) to store NPX context in the former owner's task database.
4. Record the current task as the owner of the NPX.
5. Use the FRSTOR ESC instruction (PL/M-286 RESTORE$REAL$STATUS) to load the NPX context from the new owner's TDB.

Since task switches may occur during execution of the exception handler, steps 3, 4, and 5 are a critical region and must be protected by a mechanism such as a semaphore.

The exception handler must run at PL 0, both because it alters the critical task database at PL 0 and because it uses the privileged instruction CLTS.

The exception handler must be an interrupt procedure, not an interrupt task. If it were an interrupt task, the task switch that occurs upon returning from the exception handler would set TS, thereby causing the exception again.

The return link points to the first byte of the interrupted instruction. Return from the exception handler causes restart of that instruction, but this time TS is reset and the instruction can proceed.

Interrupt 9—Processor Extension Segment Overrun (MP)

This exception occurs when a memory operand of an 80287 instruction has a segment-limit violation. Since the 80287 executes in parallel with the 80286, two difficulties may arise:

- The occurrence of this exception may not relate directly to the instruction stream being executed by the current task. A task switch may have occurred since the 80287 began executing the instruction. Even if the interrupted task is the correct task, its IP may have been advanced by several instructions beyond the ESC instruction.
- Since the exception is not maskable, it may occur while interrupts are disabled. If minimum interrupt latency is important, the exception handler must do as little as possible. It could, for example, record the error for later handling.

The offending ESC instruction cannot be restarted. The task containing the ESC instruction (which may not be the current task) must eventually be cancelled.

The exception handler must execute an FINIT instruction before executing a WAIT or other ESC instruction; otherwise, the WAIT or ESC instruction will never finish. FINIT does not affect the CS:IP value and data address saved in the 80287.

Note that the 80286 CPU detects some addressing violations before sending the ESC instruction to the 80287 NPX. In these cases, the CPU causes trap 13 (pushing an error code of zero), and it is generally possible to restart the ESC instruction. Refer to Chapter 7 for more information regarding trap 13.

Interrupt 16—Processor Extension Error (MF)

The 80287 detects six different exception conditions during instruction execution. If the detected exception is not masked by a bit in the control word, the 80287 communicates the fact that an error occurred to the CPU by a signal at the $\overline{\text{ERROR}}$ pin. The CPU causes interrupt 16 the next time it checks the $\overline{\text{ERROR}}$ pin, which is only at the beginning of a subsequent WAIT or certain ESC instructions. If the exception is masked, the 80287 handles the exception according to on-board logic; it does not assert the $\overline{\text{ERROR}}$ pin in this case.

The six exception conditions are

1. INVALID OPERATION
2. OVERFLOW
3. ZERO DIVISOR
4. UNDERFLOW
5. DENORMALIZED OPERAND
6. PRECISION (INEXACT RESULT)

The steps to be taken to remove the error condition depend on the application.

Once the exception handler corrects the error condition causing the exception, the floating point instruction that caused the exception can be restarted, if appropriate. This cannot be accomplished by IRET, however, because the trap occurs at the ESC or WAIT instruction following the offending ESC instruction. The handler must obtain from the 80287 the address of the offending instruction in the task that initiated it, make a copy of it, execute the copy in the context of the offending task, and then return via IRET to the current CPU instruction stream.

The ESC instructions that do not cause automatic checking of the $\overline{\text{ERROR}}$ pin are FNCLEX, FNINIT, FSAVE, FSETPM, FSTCW, FSTENV, and FSTSW. You can use the WAIT instruction to test the $\overline{\text{ERROR}}$ pin before these instructions, if necessary.

The offending ESC instruction cannot be restarted. The task containing the ESC instruction (which may not be the current task) must eventually be cancelled.

The exception handler must execute an FINIT instruction before executing a WAIT or other ESC instruction; otherwise, the WAIT or ESC instruction will never finish. FINIT does not affect the CS:IP value and data address saved in the 80287.

Note that the 80286 CPU detects some addressing violations before sending the ESC instruction to the 80287 NPX. In these cases, the CPU causes trap 13 (pushing an error code of zero), and it is generally possible to restart the ESC instruction. Refer to Chapter 7 for more information regarding trap 13.

Interrupt 16—Processor Extension Error (MF)

The 80287 detects six different exception conditions during instruction execution. If the detected exception is not masked by a bit in the control word, the 80287 communicates the fact that an error occurred to the CPU by a signal at the $\overline{\text{ERROR}}$ pin. The CPU causes interrupt 16 the next time it checks the $\overline{\text{ERROR}}$ pin, which is only at the beginning of a subsequent WAIT or certain ESC instructions. If the exception is masked, the 80287 handles the exception according to on-board logic; it does not assert the $\overline{\text{ERROR}}$ pin in this case.

The six exception conditions are

1. INVALID OPERATION
2. OVERFLOW
3. ZERO DIVISOR
4. UNDERFLOW
5. DENORMALIZED OPERAND
6. PRECISION (INEXACT RESULT)

The steps to be taken to remove the error condition depend on the application.

Once the exception handler corrects the error condition causing the exception, the floating point instruction that caused the exception can be restarted, if appropriate. This cannot be accomplished by IRET, however, because the trap occurs at the ESC or WAIT instruction following the offending ESC instruction. The handler must obtain from the 80287 the address of the offending instruction in the task that initiated it, make a copy of it, execute the copy in the context of the offending task, and then return via IRET to the current CPU instruction stream.

The ESC instructions that do not cause automatic checking of the $\overline{\text{ERROR}}$ pin are FNCLEX, FNINIT, FSAVE, FSETPM, FSTCW, FSTENV, and FSTSW. You can use the WAIT instruction to test the $\overline{\text{ERROR}}$ pin before these instructions, if necessary.

Extended Protection 13

CHAPTER 13
EXTENDED PROTECTION

Even though the iAPX 286 architecture provides extensive, automatic protection, a fully protected system requires additional protection features in the operating system. Operating-system software can increase the reliability of the system by providing any of these protection features:

- Extending the "type" concept
- Validating pointer parameters
- Defining the right to use operating-system objects
- Defining the right to delete segments
- Protecting shared objects that are being constructed

EXTENDED TYPE

It is both convenient and dangerous to use a selector as the name of an operating-system object. The danger arises from the fact that the selector by itself carries no information regarding the *type* of object it identifies. A program can, for example, mistakenly pass the selector of a semaphore to an operating-system procedure that operates on mailboxes, producing catastrophic results. The solution is to associate a type extension code for the object with the segment in which it resides or with a descriptor to that segment. Operating-system procedures can then check the type of objects identified by selector parameters. (The term *type extension code* is used here to avoid confusion with the processor-recognized type code in a descriptor.)

There are three general methods of associating type with operating-system objects:

1. Place the type extension code in the segment in which the object resides.
2. Associate the type code with a descriptor for the segment.
3. Use indirect names and associate the type code with the name of the object.

Only segments likely to contain named operating-system objects need to have type extension codes; namely, privilege-level 0 (PL 0), expand up, writable, data segments.

Type Extension Code with Descriptor

A logical way to store a type extension code is to associate it with the descriptor for the named object (you can view the type extension code as a refinement of the processor-recognized type code in the access-rights byte of the descriptor). The type extension code can be put in a table parallel to the descriptor table (as illustrated in Chapter 5 in connection with alias-list pointers).

Type Extension Code in Segment

It is also possible to place the type extension code at some reserved location in the data segment itself. This approach does not reference another segment and thereby avoids loading a segment register.

Indirect Naming

The most general approach to naming avoids giving to less privileged procedures *any* direct links to the named objects. Instead, names are indexes (or perhaps pointers) into a name table, which is administered by the operating system at a highly privileged level. Each entry in the name table holds the type extension code of the object, the selector of the segment in which the object resides, and any other information needed to ensure appropriate use of the object. This approach not only offers the greatest potential for protection, but also makes it possible to change the naming scheme without affecting procedures that use the names and provides a consistent way of naming both those objects that reside in dedicated segments and those that are packed into a segment with other objects.

PARAMETER VALIDATION

There is one type of privilege violation that the iAPX 286 cannot automatically check for. Consider, for example, procedure A at PL 3 that passes a pointer parameter via the stack to procedure B at PL 1. Procedure A could (accidently or purposely) pass a pointer that refers to a data structure at PL 2. Doing so would violate the intent of the protection features of the iAPX 286 because procedure A does not have sufficient privilege to operate on the data structure. However, the processor does not detect the violation because procedure B, which actually addresses the data structure, does have sufficient privilege to do so.

The iAPX 286 provides the RPL field in the selector as well as the instructions shown in table 13-1 to help software guard against such protection violations.

In addition to type checking as mentioned previously, an operating system can provide two levels of parameter validation:

1. Defensive use of ARPL instruction
2. Point-of-entry scrutiny

Defensive Use of ARPL

Simply by applying the ARPL instruction to every pointer parameter it receives, an operating system procedure guards against complicity in accessing a segment that the calling procedure has no right to access. ARPL has two selector operands, for example:

```
ARPL  sel_a, sel_b
```

Table 13-1. Access Checking Instructions

ASM286 Mnemonic	PL/M-286 Built-In Function	Description
ARPL	ADJUST$RPL	Adjust requested privilege level
VERR	SEGMENT$READABLE	Verify segment for reading
VERW	SEGMENT$WRITABLE	Verify segment for writing
LAR	GET$ACCESS$RIGHTS	Load access rights
LSL	GET$SEGMENT$LIMIT	Load segment limit

ARPL adjusts the RPL field of *sel_a* to the *greater* of its current value and the value of the RPL field in *sel_b*.

The CPL of the calling procedure is stored in the RPL field of the return pointer on the stack, as figure 13-1 illustrates. Assuming the parameter is also on the stack, statements of the form

```
MOV   AX, STACK_FRAME.RETURN_SEL
ARPL  STACK_FRAME.PARAM_SEL, AX
```

can be used to ensure that the RPL field of the parameter is not less that the calling procedure's CPL. When the called procedure uses the parameter, the processor evaluates its right to use the parameter as if it had the privilege level of the calling procedure. If the calling procedure passes a parameter that it has no right to use, an exception will occur when the called procedure uses the parameter.

Every operating system procedure should apply the ARPL instruction to every pointer or selector parameter it receives, even if the calling procedure is another operating-system procedure. This provides inexpensive protection against accidental use of invalid parameters.

Point-of-Entry Scrutiny

While use of ARPL alone is sufficient to detect such invalid parameters, it has one drawback: it does not help to isolate the source of the invalid parameter. An exception will eventually occur when some higher-level procedure uses the parameter, but this may not happen until several instructions after the procedure was called, and may not happen until after the called procedure passes the parameter to yet

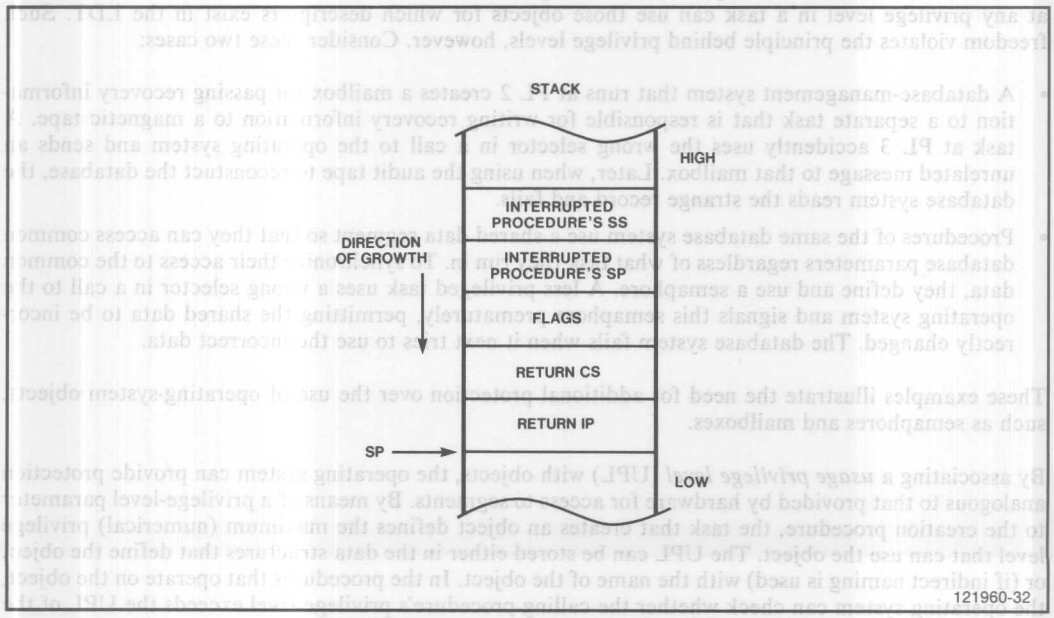


Figure 13-1. Caller's CPL

another procedure. To detect parameter errors at the earliest opportunity, the operating system should examine pointer parameters with the VERR, VERW, LAR, and LSL instructions.

The strategy for scrutinizing a pointer parameter includes

- Using ARPL as described previously to ensure that the RPL field of the pointer parameter contains the calling procedure's privilege level.
- Using VERR or VERW to ensure that the indicated segment is accessible at the calling procedure's privilege level. VERR also determines whether the indicated segment is readable; an execute-only segment, for example, is not readable. VERW also determines whether the segment is writable; only a writable data segment passes this test.
- Using LAR and LSL to make sure that the offset portion of the pointer parameter actually points to a location within the boundaries of the segment. LAR makes the access-rights byte of the indicated descriptor available, so you can determine whether the segment is an expand-down data segment. LSL makes the segment-limit field of the descriptor available. If the segment is an expand-down data segment, the offset portion of the pointer parameter must be greater than or equal to the segment limit; otherwise the offset must be strictly less than the limit.

Refer to the appropriate language reference manual for details concerning the use of these instructions.

This strategy for parameter validation is somewhat more costly than using the ARPL instruction alone, as described in the previous section. Therefore, you may wish to limit use of this strategy to those operating system procedures that can be called by less privileged, applications procedures.

USAGE PRIVILEGE LEVEL

Generally, operating system primitives that act on operating system objects (such as the semaphores and mailboxes discussed in Chapter 5) have call gates at PL 3. Without further protection, procedures at any privilege level in a task can use those objects for which descriptors exist in the LDT. Such freedom violates the principle behind privilege levels, however. Consider these two cases:

- A database-management system that runs at PL 2 creates a mailbox for passing recovery information to a separate task that is responsible for writing recovery information to a magnetic tape. A task at PL 3 accidently uses the wrong selector in a call to the operating system and sends an unrelated message to that mailbox. Later, when using the audit tape to reconstuct the database, the database system reads the strange record and fails.
- Procedures of the same database system use a shared data segment so that they can access common database parameters regardless of what task they run in. To synchronize their access to the common data, they define and use a semaphore. A less privileged task uses a wrong selector in a call to the operating system and signals this semaphore prematurely, permitting the shared data to be incorrectly changed. The database system fails when it next tries to use the incorrect data.

These examples illustrate the need for additional protection over the use of operating-system objects, such as semaphores and mailboxes.

By associating a *usage privilege level* (UPL) with objects, the operating system can provide protection analogous to that provided by hardware for access to segments. By means of a privilege-level parameter to the creation procedure, the task that creates an object defines the maximum (numerical) privilege level that can use the object. The UPL can be stored either in the data structures that define the object or (if indirect naming is used) with the name of the object. In the procedures that operate on the object, the operating system can check whether the calling procedure's privilege level exceeds the UPL of the object. The calling procedure's privilege level is readily available on the stack, as figure 13-1 illustrates.

SEND PRIVILEGE LEVEL

Moving or deleting the descriptor for a segment or the name of an object may have even more drastic effects on other tasks than using the code or structures in the segment or object. Consider the effect of deleting or mailing away a descriptor for a global data segment (for example, a translation table) shared by all tasks in the system. A task that assumes the existence of the global segment will cause an exception when it references the deleted descriptor slot. This points out the need for control over the right to send or delete a descriptor.

One way of implementing such control is to associate with each descriptor (including alias descriptors) a *send privilege level* (SPL). Procedures that move or delete descriptors (such as SEND_MESSAGE and DELETE_ALIAS) interpret the SPL and ensure that the calling procedure cannot delete (send) a descriptor from the GDT or its LDT unless CPL $<=$ SPL. SPL for segments is an attribute of segment descriptors and can be stored in tables parallel to descriptor tables.

The SPL and UPL for operating system objects may be different. In general, the SPL of all descriptors in the GDT should be zero or one, limiting the deletion and movement of GDT descriptors to the most privileged levels of the operating system. It is quite reasonable, however, to have GDT-based objects with UPL of three, so that they are accessible from any level.

CONSTRUCTING SHARED OBJECTS

Since the procedure that builds a GDT-based object, such as a mailbox or semaphore, must have a descriptor for the object's data segment, there is a possibility (however slight) that some other task (for example, an interrupt task) might mistakenly use a selector for the object under construction in a request to the operating system to use a similar object. The results are unpredictable, but probably disastrous. There are several possible methods for guarding against such a circumstance:

- Lock out all other activity for that type of object by using a semaphore or other synchronization primitive.
- For the purposes of construction only, use a reserved descriptor slot that other operating-system procedures recognize as invalid.
- For the purposes of construction only, use an LDT slot of the task that is building the object.
- Use an invalid type extension code for the object until it is completely built.

SEND PRIVILEGE LEVEL

Moving or deleting the descriptor for a segment or the name of an object may have even more drastic effects on other tasks than using the code or structures in the segment or object. Consider the effect of deleting or mailing away a descriptor for a global data segment (for example, a translation table) shared by all tasks in the system. A task that assumes the existence of the global segment will cause an exception when it references the deleted descriptor slot. This points out the need for control over the right to send or delete a descriptor.

One way of implementing such control is to associate with each descriptor (including alias descriptors) a send privilege level (SPL). Procedures that move or delete descriptors (such as SEND_MESSAGE and DELETE_ALIAS) interpret the SPL and ensure that the calling procedure cannot delete (send) a descriptor from the GDT or its LDT unless CPL <= SPL. SPL for segments is an attribute of segment descriptors and can be stored in tables parallel to descriptor tables.

The SPL and UPL for operating system objects may be different. In general, the SPL of all descriptors in the GDT should be zero or one, limiting the deletion and movement of GDT descriptors to the most privileged levels of the operating system. It is quite reasonable, however, to have GDT-based objects with UPL of three, so that they are accessible from any level.

CONSTRUCTING SHARED OBJECTS

Since the procedure that builds a GDT-based object, such as a mailbox or semaphore, must have a descriptor for the object's data segment, there is a possibility (however slight) that some other task (for example, an interrupt task) might mistakenly use a selector for the object under construction in a request to the operating system to use a similar object. The results are unpredictable, but probably disastrous. There are several possible methods for guarding against such a circumstance:

- Lock out all other activity for that type of object by using a semaphore or other synchronization primitive.
- For the purposes of construction only, use a reserved descriptor slot that other operating-system procedures recognize as invalid.
- For the purposes of construction only, use an LDT slot of the task that is building the object.
- Use an invalid type extension code for the object until it is completely built.

Glossary

Glossary

GLOSSARY

8254 Programmable Interval Timer: an Intel counter/timer device that provides three independent 16-bit counters and six counter modes. For operating-system applications the 8254 can provide timed interrupts for use by the software scheduler.

8259A Programmable Interrupt Controller: an Intel device that handles up to eight vectored priority interrupts for the CPU. The chip is designed to minimize software and real-time overhead in handling multi-level priority interrupts. It has several software selectable modes, permitting optimization for a variety of system requirements.

80286: the CPU chip of the iAPX 286 architecture.

80287: the numerics processor extension chip of the iAPX 286/20 chip set.

access rights: the attributes of a segment, defined by a descriptor, that control how a segment can be used by instructions in other segments.

accessed bit: a Boolean in the access rights byte of a descriptor that the processor sets when it loads the descriptor into a segment register.

address mapper: a hardware device that selectively translates the CPU's addressing signals into signals of another form.

alias: one of several descriptors for a segment. Each alias may define a different type, access rights, or (in some cases) limit for the segment.

alias list: a data structure that enables the operating system to find all the aliases for a given segment.

asynchronous: characterized by unpredictable order in the occurrence of events; not synchronous.

back link: the selector field in a TSS that identifies the task to be invoked when the current task executes an IRET instruction. The processor reads the back link only if the NT flag is set. The processor sets the back link to point to the TSS of the former task when a CALL instruction or interrupt causes a task switch.

base address: the 24-bit address in physical memory at which a segment starts.

busy task: either the currently executing task or a task on a back-link chain from the currently executing task. A busy task has a type code of three in the descriptor for its TSS.

Binder: see *iAPX 286 Binder.*

binding: the process of translating a symbolic reference to a form that the processor can interpret directly.

bootloader: see *bootstrap loader.*

bootstrap loader: a small, usually ROM-resident, program whose function is to load a larger, fully-featured loader.

bootloadable module: a module containing absolute object code in a simple format that expedites loading by a bootstrap loader.

boundary tag: a field at the low or high end of a block of memory used by the memory allocation algorithm to distinguish between allocated and unallocated blocks.

breakpoint: a position in a program marked in such a way that intervention can occur when execution of the program reaches that position.

buddy system: a memory-management algorithm that partitions memory into pairs of memory blocks of equal size. When both blocks of a pair are not used, they are combined into a larger block that also has a partner or "buddy" of the larger size.

buffer: an area of RAM used for transferring data to or from an external device.

built-in: in PL/M-286, a predefined identifier.

Builder: see *iAPX 286 System Builder*.

$\overline{\text{BUSY}}$: an input pin to the 80286 used by a processor extension such as the 80287 to indicate when the processor extension is unable to accept a new instruction.

call gate: a gate used to transfer control to a procedure in a segment of the same task at an equal or (numerically) lesser privilege level.

carry flag (CF): one of the six arithmetic flags typically used for unsigned integer comparisons and extended precision arithmetic. CF is set by a carry into, or a borrow from, the high-order bit of a result operand.

code segment: see *executable segment*.

combine type: one of the characteristics that the Binder associates with segments. The Binder combines segments only if they have compatible combine types.

compaction: relocating allocated memory segments into consecutive locations in order to bring together all unallocated memory blocks.

compiler control statement: source statements that specify options to the compiler. Control statements begin with a dollar sign ($) in the left margin.

conforming segment: an executable segment that executes at the CPL of any segment that calls it. A conforming segment is identified by a bit in the access rights byte of its descriptor.

control flow transfer: any change in the normal sequential progress of a program. JMP, CALL, RET, IRET, and INT instructions, as well as exceptions and external interrupts, can cause a change in control flow.

coroutine: a type of subroutine that cooperates with other coroutines in quasi-parallel execution. A set of related coroutines transfer control from one to another. Each coroutine maintains local variables across invocations and maintains an independent instruction counter that determines where to begin execution upon next invocation.

critical section: a procedure or portion of a procedure that operates on shared data in such a way that it may act incorrectly if another procedure operates on the same data within the same time interval.

CS (code segment) register: the segment register that provides addressability for access to instructions.

current privilege level (CPL): the privilege level that the processor is using to execute the currently-accessible executable segment. CPL may be equal to DPL of the executable segment, or CPL may be numerically greater that DPL if the segment is conforming.

data segment: a segment that contains data (other than immediate data) for an executable segment. A data segment is identified by a specific type code in the descriptor of the segment.

descriptor: an eight-byte item that defines the use of memory in an iAPX 286 protected-mode system.

descriptor table: one of the processor-recognized tables that contain the descriptors for the system. These tables are the GDT, the IDT, and LDTs.

descriptor privilege level (DPL): the privilege level defined in the descriptor of a segment.

device driver: the task or procedures that use knowledge of the physical characteristics of an I/O device to carry out higher-level I/O requests.

direct I/O: I/O operations in which the CPU participates.

dispatching: determining which task the processor should work on, and switching the processor to that task; also known as "scheduling."

double fault: a fault that occurs while the processor is attempting to handle a prior fault.

DS (data segment) register: one of three segment registers that provide addressability to data segments.

dynamic system: an application in which tasks begin and end relatively frequently.

EM (emulation mode) bit: a Boolean in the MSW that indicates to the processor whether ESC instruction processing is being emulated by software.

emulation: software interpretation of instructions for a processing device (such as the 80287 Numerics Processor Extension) that is not present in the system.

entry point: an executable-segment offset that identifies the starting point for execution, as when the segment is invoked via a gate.

error code: a word automatically pushed on the stack as a result of certain exceptions. The error code helps identify the segment involved in the exception.

$\overline{\text{ERROR}}$: an input pin of the 80286 used by a processor extension (such as the 80287) to signal error conditions.

ES (extra segment) register: one of three segment registers that provide addressability to data segments.

ESC or ESCAPE instruction: an instruction (usually for a processor extension) identified by the five-bit prefix 11011B.

EX (external) bit: a Boolean in the error code which, when set, indicates that the exception is due to factors outside the control of the task in which the exception occurs.

exception: a processor-detected condition that requires software intervention. The iAPX 286 communicates exceptions to software by means of the interrupt mechanism.

executable segment: a segment that contains processor instructions. An executable segment is identified by a specific type code in its descriptor.

execute-only segment: a special case of an executable segment that the processor can access for the purpose of fetching instructions but cannot access for the purposes of reading data. An execute-only segment is identified by a Boolean in the access-rights byte of its descriptor.

expand-down segment: a data segment that contains a data structure (such as a stack) that grows toward lower memory locations. An expand-down segment is identified by a Boolean in the access-rights byte of its descriptor. Offset addresses in an expand-down segment extend from the value contained in the limit field of the descriptor thru 0FFFFH.

exportation: a process by which a system-software interface is made available to applications and other system programs. Gates and segments providing access to system services (such as operating-system primitives) are placed, via the Builder's export definition, into a linkable module. This module is used when building loadable tasks that depend on the exported services.

EXPORTS list: a clause of PL/M-286's extended segmentation control syntax that specifies the public identifiers that may be referenced from outside a subsystem.

external reference: a reference to an identifier that is defined as PUBLIC in another module.

fault: an interrupt that results from an exception.

fetch policy: the algorithm that determines which segment to bring into RAM from secondary storage and when to bring it in.

first fit algorithm: a dynamic storage allocation algorithm that satisfies a request for space with the first unallocated block of storage whose size is greater than or equal to the requested size.

flag: one of several Booleans maintained by the CPU, including the arithmetic flags (CF, PF, AF, ZF, SF, OF), the control flags (TF, IF, DF), and the nested task flag (NT).

flag word: a 16-bit register of the 80286 that contains the arithmetic flags, the control flags, the nested task flag, and the IOPL. The processor saves the flag word in the TSS with each task switch and loads the flag word from the TSS of the next task, thereby enabling each task to use the flags without interference from other tasks.

fragmentation: a condition resulting from some dynamic storage-allocation algorithms, in which unallocated storage is dispersed in many small areas.

gate: a gate descriptor.

gate descriptor: a descriptor that defines a protected entry point to an executable segment or task.

GDT register: a register of the 80286 that contains the base address and limit of the GDT.

global descriptor table (GDT): the descriptor table that contains descriptors that can be used by every task in the system. There is only one GDT per processor.

handler table: a table of selectors to call gates that identify the procedures for servicing asynchronous events such as interrupts or software signals. A handler table is used by an operating system's interrupt distributor and signalling primitives.

I bit: a Boolean in the error code which, when set, indicates that index portion of the error code points to an entry in the IDT.

iAPX 286 Binder: an iAPX 286 program development utility used to link modules, combine segments, and create a single-task, loadable output module.

iAPX 286 System Builder: the configuration utility for iAPX 286 protected-mode systems.

IDT register: an 80286 register that stores the base address and limit of the IDT.

index: the field of a selector that identifies a slot in a descriptor table.

indirect I/O: a style of I/O interface in which I/O operations are executed by an independent processor, not by the CPU.

interrupt: 1) the electrical or logical signal that an event has occurred; 2) the mechanism by which a computer system responds quickly to events that occur at unpredictable times.

interrupt controller: a device (such as Intel's 8259A Programmable Interrupt Controller) that assists the CPU in responding to multiple external interrupt signals by performing such functions as detection, priority resolution, and identification.

interrupt descriptor table (IDT): a descriptor table that contains gates to the handler procedures or handler tasks for interrupts and traps. The IDT may contain only interrupt gates, trap gates, and task gates.

interrupt distributor: an operating-system interrupt procedure that transfers control to a task-defined procedure for servicing the interrupt.

interrupt-enable flag (IF): a control flag of the 80286 that determines whether the processor responds to external interrupt signals presented at the processor's INTR pin.

interrupt gate: a gate that identifies the entry point of a procedure for handling an interrupt. When an interrupt transfers control through an interrupt gate, the processor resets the interrupt-enable flag. Interrupt gates are valid only in the IDT.

interrupt handler: a procedure or task that is invoked by an interrupt.

interrupt latency: the time from the occurrence of an interrupt signal to the execution of the first instruction of an interrupt handler.

interrupt procedure: an interrupt handler that is identified by an interrupt gate or trap gate. An interrupt procedure runs in the interrupted task.

interrupt task: an interrupt handler that is identified by a task gate and runs as a task separate from the interrupted task.

interrupt vectoring: the mapping from an interrupt source to the interrupt handler. In the iAPX 286 architecture, the 8259A and the IDT are components of the interrupt vectoring process.

intersegment reference: a reference to a location in a segment other than the segment containing the reference.

intrasegment reference: a reference to a location in the same segment as the segment containing the reference.

interlevel reference: an intersegment reference to a segment that has a different privilege level than that of the segment containing the reference.

intertask transfer: a transfer of control flow to a task other that the current task.

I/O-mapped I/O: a style of I/O interface in which I/O devices respond to addresses in an address space that is distinct from the memory address space. Special I/O instructions (IN, INS, OUT, OUTS) trigger I/O operations. The 80286 uses the M/$\overline{IO}$ pin to distinguish memory addresses from I/O addresses.

I/O privilege level (IOPL): a two-bit item in the flag register of the 80286 that controls the current task's right to execute the I/O-related instructions IN, INS, OUT, OUTS, CLI, STI, LOCK. A procedure may not execute any of these instructions if CPL>IOPL.

I/O subsystem: the portion of an operating system that deals with filing and I/O.

IP (instruction pointer): an 80286 register that contains the offset of the instruction to be executed within the current code segment.

kernel: that portion of an operating system that implements the most primitive of its functions.

LDT register: an 80286 register that stores the selector, base address, and limit of the current LDT.

limit: the field of a descriptor that defines the offset of the last byte of the segment.

linkable module: an object module created by iAPX 286 translators, by the Binder, or by the Builder that can serve as input to either the Builder or the Binder. A linkable module requires further processing before it can be executed.

linking: the process of combining segments from one or more input modules and resolving references between modules. The Binder provides linking services for iAPX 286 program development.

loadable module: an executable object module (usually created by the Builder or the Binder) that has a format suitable for processing by a loader running under control of an operating system.

loader: the task or procedures of an operating system that places an object module in RAM and prepares it for execution by the operating system and the processor.

load-time: at the time a task is loaded.

local descriptor table (LDT): the descriptor table that contains descriptors that are (generally) private to a given task. Each task may have an LDT. A task can use only descriptors that are in its LDT or in the GDT. The LDT protects tasks from one another.

logical segment: the representation of a segment used by translators and program development utilities prior to the time when the segment is actually placed in physical memory.

machine status word (MSW): an 80286 register that includes the TS, EM, MP, and PE Booleans. These items are global to the system and are not a part of the task state.

mailbox: a software mechanism for sending messages between tasks. The mechanism consists of two queues: one a queue of undelivered messages, the other a queue of tasks waiting for messages. Messages are delivered in FIFO order.

memory management: the set of operating system functions that deal with the allocation of RAM to tasks.

memory-mapped I/O: a style of I/O interface in which I/O devices respond to specific addresses in the memory address space. I/O operations are triggered by standard instructions that read from, and write to, memory locations.

message: a unit of intertask communication.

module: a compilation unit or a combination of compilation units.

MP (math present) flag: a Boolean in the MSW that indicates whether a processor extension (such as the 80287 Numerics Processor Extension) is present.

mutual exclusion: preventing two critical sections from executing concurrently.

multiprocessing: using more than one CPU to execute a multitasking system.

multitasking: the capability to support more than one task either simultaneously (by using more than one CPU) or virtually simultaneously (by multiplexing one CPU among several tasks).

nested task (NT) flag: a Boolean in the flag word that indicates the existence of a back link field in the TSS to a previous TSS.

non-maskable interrupt (NMI): an external interrupt presented to the NMI pin of the 80286 that the processor does not ignore, even when IF is reset.

not-present segment: a segment whose descriptor has the present bit reset. In a virtual-memory system, this condition normally indicates that the segment has been evicted from RAM to make space for other segments.

nucleus: see *kernel*.

nullify: assign a null value to.

numerics processor extension (NPX): the 80287 processor which cooperates with the 80286 CPU to extend its processing power in mathematical applications.

object module format (OMF): a standard for the structure of object code files.

OF (overflow) flag: an arithmetic flag that indicates when a signed operation produces a positive number that is too large or a negative number that is too small to fit in the destination operand.

offset: the address of a location within a segment, expressed as a quantity to be added to the base address of the segment.

outward call: the attempt to call a procedure in another segment whose DPL is numerically greater than CPL.

parallel table: a table whose entries have a one-to-one correspondence with the entries of a descriptor table, used to associate additional information with each descriptor.

parallelism: concurrent execution of two or more tasks or devices.

parameter validation: checking the attributes of parameters passed between procedures at different privilege levels to prevent protection violations or to detect exception conditions.

PE bit: a Boolean in the MSW that indicates whether the 80286 is running in real-address mode or in protected, virtual-address mode.

Petri net graph: a notation for visualizing Petri nets. Petri nets are a mathematical tool for modeling systems, first proposed by Dr. Carl Adam Petri in 1962.

physical address: a 24-bit address, such as that used as a base address, capable of encompassing the entire address space of the 80286.

physical segment: a segment as viewed by the processor, to be distinguished from "logical segment."

PIC: programmable interrupt controller. See *interrupt controller*.

pipes: a mechanism for intertask communication used in the UNIX operating system. Each task views a communication channel as a file and uses READ and WRITE operations to receive and send messages.

placement policy: the algorithm for determining where in RAM to locate a segment.

pointer: an item that specifies a memory location. A full or long pointer includes a selector, which indirectly chooses a segment base address, and an offset value, which points to a specific address within that segment. An offset by itself is called a *short* pointer.

preemption: a dispatching process in which the operating system switches to another task even though the current task has not requested any function that would cause it to wait. The operating system may preempt one task in order to give other tasks a share of CPU attention.

prefix: one of several instruction codes that modify the function or the environment of the following instruction. iAPX 286 prefixes include the LOCK prefix, repeat prefixes, segment-override prefixes, and the ESCAPE prefix.

present bit: a Boolean in a segment descriptor that indicates whether the segment is actuallly present in RAM. In a virtual-memory system, the segment may have been evicted from RAM to create space for another segment.

primitive: one of the operating-system operations made accessible to applications by some explicit mechanism. In the iAPX 286 architecture, primitives are typically procedures with call gates in the GDT or LDTs.

privilege: the right to access certain portions of memory or to execute certain processor instructions.

privilege level (PL): a measure of privilege. In the iAPX 286 architecture, privilege is measured by integers in the range 0–3, where 0 is the most privileged and 3 the least.

privileged instruction: an instruction that can be executed only by a procedure running at privilege level 0. Privileged instructions include CLTS, HLT, LGDT, LIDT, LLDT, LMSW, and LTR.

processor extension: an optional, special-purpose processor (such as the 80287) that runs in parallel with the 80286 and extends its processing power.

profiler: a procedure or task that collects data about segment usage.

protected, virtual-address mode: the mode of operation of the 80286 that provides virtual-memory addressing and memory protection.

protection: a mechanism that limits or prevents access to areas of memory or to instructions.

public: a symbol available for intermodule reference.

readable segment: an executable segment that can be read. It is necessary to read from an executable segment if that segment contains constants. A readable segment is identified by a Boolean in the access rights byte of its descriptor.

read-only segment: a data segment that cannot be written to. A read-only segment is identified by a Boolean in the access rights byte of its descriptor.

real-address mode: the mode of operation of the 80286 that provides greatest compatability with the 8086, without protection and virtual memory addressing.

real memory: the physical memory, as distinguished from virtual memory.

real-time system: a system that responds to external events in a relatively short time, as contrasted with a batch system.

region: a mechanism for providing mutual exclusion among critical sections. A region is similar to a semaphore that has the additional properties that 1) only the task that acquires a region can release it, and 2) a task cannot be suspended while holding a region.

relocation: changing the physical location of a segment.

replacement policy: the algorithm that determines when to remove segments from RAM and which segments to remove when space is (or is likely to be) needed for another segment.

resolving reference: see *binding*.

requested privilege level (RPL): the privilege-level field of a selector. A procedure may request a numerically greater privilege level for use of a segment by placing the desired privilege level in the RPL field of the selector that identifies that segment.

run-time: the time a task executes.

scheduler queue segment: a segment that contains one or more of the task queues used by a scheduler. Keeping a queue in one segment may reduce the time for searching the queue.

scheduling: see *dispatching*.

scheduling mode: one of two styles of scheduling a task: hardware- (interrupt-) scheduled mode or software-scheduled mode.

scheduling state: one of several conditions that affect the way a task is treated by the scheduler. A task may be ready to execute, executing, waiting for some event, etc.

secondary storage: a slower, less expensive storage medium than RAM (for example, disk).

segment: a variable-length area of contiguous memory addresses not exceeding 64K bytes.

segment register: one of four 80286 registers that hold addressing information for the segments that are currently addressable by a task. The segment registers are CS (code segment), DS (data segment), ES (extra segment), and SS (stack segment).

selector: an item that identifies a descriptor by the location of the descriptor in a descriptor table.

semaphore: a synchronization mechanism that communicates the occurrence of an event between two (or more) tasks via a shared memory location.

send privilege level (SPL): a software-implemented measure of the right to send or delete a segment.

shadow task: a duplicate task used to enable the operating system to perform an outward call.

signal: a mechanism for permitting one task to communicate the occurence of an event to another task that is not waiting for the event to occur.

single step: a mode of execution that permits intervention between each instruction; used primarily as a debugging aid.

slot: an entry in a descriptor table.

SS (stack segment) register: the segment register that provides addressability to the current stack segment.

stack segment: a segment used by the processor to hold return addresses, dynamic data, temporary data, and parameters. For greatest protection, each privilege level of a task may have its own stack. Stack segments usually expand downward.

static system: an application in which the mix of tasks does not change over time.

subsystem: in PL/M-286, a collection of tightly-coupled, logically-related modules that obey the same model of segmentation.

swap space: the secondary storage area used to contain segments that have been removed from RAM.

swapping: in a virtual-memory system, the process of moving segments between RAM and secondary storage.

swapping manager: a procedure or task responsible for swapping.

synchronization: imposition of an order on the occurrence of certain events.

system segment: a segment containing a descriptor table or task state.

table indicator (TI): a Boolean in a selector that identifies the descriptor table to which the selector refers.

task: a single thread of execution that has an associated processor state.

task database: the collection of information about a task that the operating system needs to store.

task information block: a segment or part or a segment used by the operating system to contain all or part of a task database.

task gate: a gate that identifies a TSS. A control transfer through a task gate causes a task switch.

task register: an 80286 register that points to the TSS of the currently active task.

task state segment (TSS): a segment used by the processor to store the contents of the task-variable registers, the stack-segment selectors and pointers for the three most privileged levels, the selector for the task's local descriptor table, and a back link that may point to another task in a chain of nested task invocations.

TF (single step flag): a Boolean in the flag word that, when set, indicates that the processor should cause a trap after each instruction. Typically, TF is used to facilitate debugging.

thrashing: in a virtual memory system, a condition in which excessive swapping seriously degrades performance of all tasks.

time-sharing system: a multi-user, multitasking system in which processors are multiplexed among users.

time slice: the time interval for which the CPU is allocated to a task.

translator: an assembler or compiler.

trap: an interrupt due to an exception condition.

trap gate: a gate that identifies the procedure to handle a trap. An interrupt through a trap gate differs from an interrupt through an interrupt gate in that, with a trap gate, interrupts are not disabled upon entry into the procedure. Trap gates are valid only in the IDT.

Trojan horse: a type of protection violation in which a procedure passes a selector that it has no right to use to a more privileged procedure that does have the right to use it.

trusted instruction: one of a set of I/O-related instructions that cannot be executed unless CPL is less than or equal to IOPL. The trusted instructions are CLI, STI, IN, INS, OUT, OUTS, and LOCK.

type code: a value in a descriptor that specifies the intended use of a segment. The processor interprets the type code to ensure that segments are used only as intended.

type extension code: a software extension to the type code concept that includes usages recognized by the operating system.

UDI (Universal Development Interface): an Intel standard for interfaces to operating-system services.

usage privilege level (UPL): a software-defined measure of the right to use an operating-system object.

vectoring: see *interrupt vectoring.*

virtual address: an address that consists of a selector and an offset value. The selector chooses a descriptor for a segment; the offset provides an index into the selected segment.

virtual address space: The set of all possible virtual addresses that a task can access, as defined by the GDT and the task's LDT. The maximum possible virtual address space for one task is one gigabyte.

virtual memory: a style of memory management that permits the virtual address space to exceed the physical address space of RAM. With the help of processor features, the operating system simulates the virtual address space by using secondary storage to hold the overflow from RAM.

word count: a field of a gate descriptor that specifies the number of words of parameters to be copied from the calling procedure's stack to the stack of the called procedure.

writable segment: a data segment that can be written to. A writable segment is identified by a Boolean in its descriptor.

XOS (Example Operating System): an imaginary operating system, portions of which are used in this book as examples.

INDEX

8089 I/O processor, 8-1
8254 Programmable Interval Timer (PIT), 4-5, 4-9, 4-13
8259A Programmable Interrupt Controller (PIC), 4-6, 6-1, 6-6, 7-6
80287 Numerics Processor Extension (NPX), 1-7, 7-4, 7-5, 7-8, Chapter 12

access rights (field of descriptor), 2-21, 2-22, 9-2, 11-5, 11-10, 13-1, 13-2, 13-4
accessed bit, 2-7, 5-3, 9-1, 9-4, 9-5, 9-7, 10-2
ADC, 7-8
addressing mechanism, 2-1
ADJUST$RPL, 2-16, 13-2
alias, 2-17 thru 2-22, 3-2, 4-10, 4-12, Chapter 5, 6-5, 8-2, 9-3, 9-7, 10-3, 11-2, 11-9, 13-5
ARPL, 2-16, 13-2 thru 13-4
ASM286, 3-6, 4-13, 11-4, 13-2
ASSUME, 11-4
asynchronous execution, 1-6

back link (of TSS), 4-1 thru 4-4, 4-7, 4-13, 7-5, 7-6, 10-2, 11-10
base address, 2-4, 2-15, 4-1, 5-3, 9-3, 9-5, 10-3, 11-3, 11-10, 11-11
binding, 1-8, 2-8, 2-9, Chapter 11
Binder, *see* iAPX 286 Binder
bootloadable, *see* module, bootloadable
BOUND, 7-4
bound check exception, 7-4
boundary tags, 3-2, 3-5, 3-6, 9-3
breakpoint, 2-17, 7-3
buffer, 2-18, 3-1, 3-10, 5-10, 8-1, 8-3, 8-5, 8-7, 9-2, 9-4
Builder, *see* iAPX 286 System Builder
BUSY/ pin, 12-2, 12-3
busy task, 4-3, 4-4, 7-7

CALL, 2-7, 2-9, 2-10, 4-2 thru 4-4, 4-13, 4-17, 6-1, 6-2, 6-4, 6-6, 7-5 thru 7-7, 11-3
carry flag (CF), 7-8
CLI, 4-6, 4-7, 5-6, 6-2, 8-2
CLTS, 12-2, 12-4
CMPS, 7-7
combine type, 11-4
COMPACT, 11-4, 11-5
compaction, 3-10
compiler control statements, 11-4
conforming segment, 2-5, 6-2, 6-7, 6-8
critical section, 5-5 thru 5-7, 12-4
CS register, 4-4, 6-9, 7-5 thru 7-7, 9-1, 9-6, 9-7, 10-1, 11-3, 11-9, 12-5
current privilege level (CPL), 2-11, 2-16, 4-6, 5-6, 6-2, 6-7, 6-8, 8-2, 10-1, 10-2, 13-3

deadlock, 5-7, 5-10
debugger, 2-17, 7-3, 11-3, 11-10
deletion
  of segment, 2-20, 5-2, 5-3, 8-3, 9-5, 13-1
  of descriptor, 13-5
DESCRP section, 11-11, 11-12
descriptor, 2-2, 2-12, 11-6, 11-9
  data segment, 2-2, 2-3, 2-6, 2-14, 2-15, 2-18, 7-7, 9-3 thru 9-5, 9-7
  dynamic creation, 3-1, 3-2
  executable segment, 2-2, 2-4, 2-6, 2-7, 2-14, 2-15, 2-17, 6-7, 7-7, 9-7
  gate, 2-2, 2-7 thru 2-11, 2-15, 6-2, 7-6, 9-1, 9-2, 10-2, 11-3, 11-9
  system segment, 2-2, 2-5
descriptor privilege level (DPL), 2-3, 2-6, 2-9, 2-11, 2-15, 2-20, 2-21, 4-3, 5-7, 6-7, 6-8, 8-2. 8-5, 8-7
descriptor table, 2-2, 2-12, 2-15
  see also global descriptor table, local descriptor table, interrupt descriptor table
DESNAM section, 11-11
device drivers, 5-17, 8-1, 8-2, 8-4, 8-7, 9-2, 9-4, 11-2, 11-4
DI register, 7-7
DISABLE, 6-2, 6-4, 7-1
dispatching, 4-9, 4-13, 5-7, 9-5, 11-4
divide error exception, 6-7, 7-3
double fault, 7-5, 9-2, 9-4
DPL, *see* descriptor privilege level
DS register, 2-17, 2-18, 2-21, 4-12, 7-5 thru 7-7, 9-1, 9-6, 9-7, 10-1
dynamic system, 1-7 thru 1-10, 2-2, 2-3, 2-12, 2-18 thru 2-20, 2-22, 3-1, 4-4, 5-7, 6-5, 11-8

effective privilege level, 2-16
EM (emulation mode) flag, 7-4, 7-8, 12-1 thru 12-3
emulation, 7-8, 12-1, 12-4
ENABLE, 6-2
ENTER, 7-6
ENTRY, 11-6
EPROM, 10-3
ERROR/ pin, 7-8, 12-1 thru 12-3, 12-5
error code, 7-1, 7-2, 7-5 thru 7-7, 7-9, 9-2, 9-3, 12-5

ES register, 2-17, 2-18, 2-21, 2-22, 4-12, 7-5 thru 7-7, 9-1, 9-6, 9-7, 10-1
ESCAPE instructions, 7-4, 12-1 thru 12-5
EX (external) bit, 7-2
examples, 2-20 thru 2-26, 3-1 thru 3-9, 3-15 thru 3-21, 4-15, 5-8, 5-9, 5-13 thru 5-16, 10-3 thru 10-15, 11-6 thru 11-8, 11-12 thru 11-28
exception, 1-7, 2-8, 2-10, 2-17, 4-3, 4-7, 5-5, 6-2, 6-4, 6-9, Chapter 7, 9-1, 11-3, 13-3, 13-5
  handler, 6-7, Chapter 7, 9-2, 9-4, 11-3, 12-4, 12-5
  recovery, Chapter 7, 12-3 thru 12-5
EXECUTE, 11-9
execute-only segment, 7-7, 13-4
expand-down segment, 2-5, 11-11, 13-4
expansion direction, 2-5, 5-3
EXPORT, 11-6
export module, 11-6, 11-12, 11-13
exports list, 11-4, 11-5
extended segmentation controls, 11-4
EXTERNAL, 3-6

FAR, 11-4
fault, *see* exception
FINIT, 12-5
"first fit" algorithm, 3-1, 3-2
flag word, 4-2 thru 4-4, 6-2, 6-4, 8-1, 8-2, 10-1
FNINIT, 12-2, 12-5
FORK, 11-9
fragmentation, 3-1, 3-2, 9-6
FRSTOR, 12-4
FSAVE, 12-4, 12-5
FSETPM, 12-2, 12-5

gate, 2-8, 2-11, 11-6
  call, 2-8, 2-10, 2-12, 2-14, 2-15, 2-21, 3-9, 4-13, 5-7, 5-12, 6-1, 6-2, 8-7, 11-3, 11-12, 11-14
  interrupt, 2-8, 2-15, 6-2, 6-4, 6-7
  name, 11-6
  task, 2-8, 2-14, 2-15, 4-3, 4-7, 6-2, 6-4, 7-5, 10-3
  trap, 2-8, 2-15, 6-2, 6-4, 6-7
  see also descriptor, gate
GATE, 11-6
GDT, *see* global descriptor table
GDT register, 2-15, 10-2
general protection exception, 7-7, 8-2
GET$ACCESS$RIGHTS, 13-2
GET$SEGMENT$LIMIT, 3-6, 13-2
global descriptor table (GDT), 1-3, 2-12, 2-14 thru 2-19, 2-22, 3-3, 3-4, 3-7, 3-8, 4-1, 4-3, 4-10 thru 4-12, 5-1, 5-7, 5-10, 6-7, 7-2, 8-7, 10-2, 10-3, 11-2, 11-4, 11-5, 11-9 thru 11-12, 13-5
handler table, 6-7
hashing algorithm, 5-3

I bit (of error code), 7-2
iAPX 286 Binder, 1-10, 11-3, 11-5, 11-6, 11-10, 11-11
iAPX 286 System Builder, 1-8 thru 1-11, 2-2, 2-4, 2-12, 2-18, 3-1, 3-6, 8-2, 10-3, 11-3 thru 11-6, 11-8, 11-10, 11-11
iAPX 386, 2-3
identifier, 5-7, 5-10, 8-7
  see also interrupt identifier
IDT, *see* interrupt descriptor table
IDT register, 2-15, 6-2
IF (interrupt-enable) flag, 4-6, 5-6, 6-2, 6-4
index field (of selector), 2-16, 2-17, 5-3, 7-2
INIT$REAL$MATH$UNIT, 12-2
initialization
  of 80287, 12-2
  see also system initialization
input/output (I/O), 1-4 thru 1-7, 2-18, 4-5, 4-7, 4-10, 5-10, Chapter 8, 9-2, 9-4, 9-8, 11-2, 11-4, 11-9
  indirect, 8-3
  memory-mapped, 8-2
IN, 8-1
INS, 7-7, 8-1
INT, 2-7, 2-10, 2-15, 4-3, 6-1, 6-2, 7-3
INTA cycles, 2-15
INTO, 6-1, 6-2, 7-3
INTR pin, 6-1, 6-2, 6-6
interrupt, 1-4, 1-7, 2-10, 2-15, 2-21, 4-3, 5-4 thru 5-6, Chapter 6, 7-1, 7-6, 7-7, 8-4, 10-1, 10-2
  distribution, 6-7 thru 6-9
  flag, *see* IF
  identifier, 2-15, 6-1, 6-2
  handler, 2-15, 2-18, 4-6
  latency, *see* interrupt response time
  mask, 4-7, 12-4
  procedure, 4-4, 4-7, 4-11, 4-13, 6-2, 6-4 thru 6-7, 7-1, 12-4
  response time, 4-3, 5-5, 6-5, 12-4
  scheduled task, *see* scheduling
  software, 6-1
  task, 4-4, 4-7, 6-2, 6-4 thru 6-6, 7-1, 7-6, 9-4, 12-4, 13-5
interrupt descriptor table (IDT), 2-12, 2-15, 2-16, 2-18, 4-3, 4-5, 4-7, 6-2, 6-4, 6-5, 6-7, 7-2, 7-5, 10-2, 10-3, 11-11

"invalid TSS" exception, 7-5 thru 7-7, 9-2, 9-5
IOPL (I/O privilege level), 4-6, 5-6, 8-1, 8-2, 8-5
IP register, 4-4, 6-9, 7-1, 7-3, 7-5, 7-7, 10-1, 11-9, 12-4, 12-5
IRET, 2-7, 2-10, 4-3, 4-4, 4-13, 6-4, 6-6, 6-7, 6-9, 7-3, 7-5, 7-7, 8-2, 12-5

JMP, 2-5, 2-7, 2-9, 2-10, 4-3, 4-4, 4-13, 7-7, 10-1, 10-3

LABEL, 11-4
LAR, 9-7, 13-2, 13-4
LARGE, 11-4
LDT, *see* local descriptor table
LDT register, 2-14, 4-3, 7-6, 9-1
LDT selector (of TSS), 4-1, 4-3, 6-9, 7-5, 10-2, 11-11
LEAVE, 7-6
LGDT, 2-15
LIDT, 2-15, 6-2
limit field (of descriptor), 2-4, 2-5, 2-12, 2-15, 2-16, 4-1, 5-3, 7-5 thru 7-7, 9-5, 10-1, 10-3, 11-11, 12-4, 13-2, 13-4
linkage, 1-10, 7-6
LLDT, 2-14, 7-6
LMSW, 10-1, 12-1
loading, *see* program loading
local descriptor table (LDT), 2-5, 2-12, 2-14, 2-16, 2-18 thru 2-20, 3-1, 4-10, 5-1 thru 5-3, 5-10, 5-17, 6-7, 7-2, 8-7, 9-3, 9-4, 9-7, 10-2, 10-3, 11-2, 11-4, 11-5, 11-8 thru 11-12
  not present, 9-1, 9-2
LOCAL$TABLE, 2-14
LOCK, 8-2
LODTXT section, 11-11, 11-12
logical segment, 1-1, 11-2 thru 11-5
LSL, 9-7, 13-2, 13-4
LTR, 4-1, 9-1

MACHINE$STATUS, 12-1
mailbox, 5-10 thru 5-12, 5-16, 5-17, 8-7, 9-3, 9-5, 11-2, 13-1, 13-4, 13-5
mechanisms, *see* policies and mechanisms
memory management
  real, 1-8, 1-9, 2-22, Chapter 3, 5-4, 5-12, 5-17, 8-3, 8-5, 11-6
  virtual, 1-8, 2-2, 2-3, 2-7, 7-6, Chapter 9, 11-9
message, 2-19, 3-9, 4-9, 5-4, 5-10, 5-12, 5-16
module, 11-1 thru 11-6
  bootloadable, 11-6, 11-10
  linkable, 11-6, 11-10
  loadable, 1-10, 11-10, 11-11, 11-14
  object, 1-8, 1-9, 11-9, 11-10
MOV, 7-6
MOVS, 7-7
MP (math present) flag, 7-4, 12-1 thru 12-4
MSW (machine status word), 7-4, 7-8, 10-1, 12-1, 12-2
multiprocessor systems, 3-10
mutual exclusion, 5-5, 5-6

naming, 11-1 thru 11-6, 11-11, 11-12, 13-1, 13-2, 13-4, 13-5
NEAR, 11-4
NMI (non-maskable interrupt), 6-1, 6-2, 10-2
NOT PRESENT statement, 11-12
"not present" exception, 7-5 thru 7-7, 9-1, 9-3 thru 9-8, 11-3
  see also present bit
NT (nested task) flag, 4-2 thru 4-4, 4-7, 7-6, 9-4, 11-10
Numerics Processor Extension (NPX), *see* 80287

OBJECT, 11-6
object module format (OMF), 11-8, 11-10, 11-11
OF (overflow) flag, 7-3
OUT, 8-2
OUTS, 7-7, 8-2
outward call, 6-8
overflow exception, 7-3

paged architecture, 9-6
parallel table, 5-3, 13-1
parallelism, 1-4, 8-4, 8-7
PE (protection enable) flag, 10-1
Petri net graph, 8-4, 8-5
physical address, 1-8, 1-9, 2-4, 3-2 thru 3-4, 11-10
physical segment, 1-1, 1-2, 2-12, 2-15, 11-1 thru 11-3
PIC, *see* 8259A Programmable Interrupt Controller
pipes, 5-17
PL/M-286, 2-14 thru 2-16, 2-21, 2-22, 3-3, 3-6, 4-1, 4-3, 4-13, 6-2, 11-4, 11-6, 12-2, 12-4, 13-2
pointer parameters, *see* selector parameters
policies and mechanisms
  scheduling, 4-9, 4-10
  virtual memory management, 9-1, 9-6
POP, 7-6
POPA, 7-7
POPF, 8-2
preemption, 4-5, 4-7, 4-9, 4-10, 4-13
prefix, 7-1

subsystem, 11-4
swapping, 5-12, 8-3, 9-2 thru 9-4, 9-8
synchronization, 2-22, 3-8, 3-9, Chapter 5, 8-4, 8-6, 12-2, 13-4, 13-5
system initialization, 6-5, 6-6, Chapter 10

TABLE, 11-6
table indicator (TI), 2-16, 2-17, 7-2
task, 3-1
creation, 2-2, 2-18, 3-1, 5-12, 6-9, 11-9
database (TDB), 4-11, 5-12, 9-4, 11-9, 11-10, 12-3, 12-4
management, Chapter 4
state, 4-1, 10-2, 12-3
switching, 2-14, 4-1, 4-3, 4-4, 4-9, 5-6, 6-2, 7-5 thru 7-7, 8-2, 9-4, 9-5, 10-2, 11-9, 12-2 thru 12-4
TASK, 11-6
task register (TR), 4-1, 4-2, 4-4, 9-1, 10-2
task state segment (TSS), 2-5, 2-14, 2-18, 3-1, 4-1 thru 4-4, 4-10 thru 4-13, 5-1, 6-2, 6-4, 6-9, 7-1, 7-5, 7-7, 8-2, 9-1, 9-4 thru 9-7, 10-2, 10-3, 10-5, 11-6, 11-8 thru 11-12, 11-14
TASKREGISTER, 4-1
TDB, see task database
termination (of task), 6-9, 7-1, 7-5, 12-5
TF, see single-step flag
TI, see table indicator
time slice, 4-5, 4-9 thru 4-11
timer, see 8254 Programmable Interval Timer
TS (task switched) flag, 7-4, 12-1 thru 12-4
TSS, see task state segment
type
field of descriptor, 2-5, 2-17, 2-18, 2-20, 2-21, 4-3, 4-4, 6-2, 9-3, 11-10, 11-14, 13-1
extended, 13-1, 13-2, 13-5

UDI (Universal Development Interface), 8-1
"undefined opcode" exception, 7-4
UNIX, 11-9
usage privilege level (UPL), 8-7, 13-4, 13-5

vectoring, 6-1, 6-3, 7-1
VERR, VERW, 13-3, 13-4
virtual memory, see memory management, virtual
WAIT, 7-4, 7-8, 12-1 thru 12-5
WAITFORSIGNAL TRAP, 4-3
word count, 2-8
writable data segment, 2-6, 9-5, 11-10, 11-14, 13-1, 13-4

XOS, 11-1, 11-2, 11-4, 11-6

subsystem, 11-4
swapping, 5-12, 8-3, 9-2 thru 9-4, 9-8
synchronization, 2-22, 3-8, 3-9, Chapter 5, 8-4, 8-6, 12-2, 13-4, 13-5
system initialization, 6-5, 6-6, Chapter 10

TABLE, 11-6
table indicator (TI), 2-16, 2-17, 7-2
task, 1-1
  creation, 2-2, 2-18, 3-1, 5-12, 6-9, 11-9
  database (TDB), 4-11, 5-12, 9-4, 11-9, 11-10, 12-3, 12-4
  management, Chapter 4
  state, 4-1, 10-2, 12-3
  switching, 2-14, 4-1, 4-3, 4-4, 4-9, 5-6, 6-2, 7-5 thru 7-7, 8-2, 9-4, 9-5, 10-2, 11-9, 12-2 thru 12-4
TASK, 11-6
task register (TR), 4-1, 4-2, 4-4, 9-1, 10-2
task state segment (TSS), 2-5, 2-14, 2-18, 3-1, 4-1 thru 4-4, 4-10 thru 4-13, 5-1, 6-2, 6-4, 6-9, 7-1, 7-5, 7-7, 8-2, 9-1, 9-4 thru 9-7, 10-2, 10-3, 10-5, 11-6, 11-8 thru 11-12, 11-14
TASK$REGISTER, 4-1
TDB, *see* task database
termination (of task), 6-9, 7-1, 7-5, 12-5
TF, *see* single-step flag
thrashing, 9-8
TI, *see* table indicator
time slice, 4-5, 4-9 thru 4-11
timer, *see* 8254 Programmable Interval Timer
TS (task switched) flag, 7-4, 12-1 thru 12-4
TSS, *see* task state segment
type
  field of descriptor, 2-5, 2-17, 2-18, 2-20, 2-21, 4-3, 4-4, 6-2, 9-3, 11-10, 11-14, 13-1
  extended, 13-1, 13-2, 13-5

UDI (Universal Development Interface), 8-1
"undefined opcode" exception, 7-4 .
UNIX, 11-9
usage privilege level (UPL), 8-7, 13-4, 13-5

vectoring, 6-1, 6-3, 7-1
VERR, VERW, 13-2, 13-4
virtual memory, *see* memory management, virtual

WAIT, 7-4, 7-8, 12-1 thru 12-5
WAITFORINTERRUPT, 4-3
word count, 2-8
writable data segment, 2-6, 9-5, 11-10, 11-14, 13-1, 13-4

XOS, 11-1, 11-2, 11-4, 11-6

present bit, 2-2, 2-3, 2-9, 5-3, 7-5, 7-6, 9-1 thru 9-5, 11-12
primitives, 11-3, 11-4, 11-6, 11-12, 11-14, 13-4
priority, 2-19, 4-6, 4-7, 4-9, 4-10, 6-6, 6-7
privilege level, 1-3, 1-4, 1-8, 2-6, 2-7 thru 2-11, 2-15, 2-18, 2-20, 2-21, 3-8, 3-9, 4-1, 4-2, 4-9, 4-12, 4-13, 5-2, 5-7, 5-10, 5-12, 6-2, 6-3, 6-5, 6-8, 8-2, 8-3, 8-7, 9-3, 9-7, 11-2, 11-6, 11-9, 11-10, 12-2, 12-4, 13-1, 13-2, 13-4, 13-5
PROC, 11-4
"processor extension error" exception, 7-8, 12-5
"processor extension not available" exception, 7-4, 12-3, 12-4
"processor extension segment overrun" exception, 7-5, 12-4, 12-5
profiling, 9-7, 9-8
program loading, 1-8, 1-10, 2-3, 2-4, 3-1, 6-5, Chapter 11
  bootstrap, 1-9, 11-10
protected, virtual-address mode, 10-1, 10-2, 12-2
protection, 1-3 thru 1-5, 1-9, Chapter 2, 3-1, 3-7, 4-3, 4-4, 4-6, 5-1, 5-4, 5-7, 5-10, 6-2, 6-3, 6-5, 6-6, 7-3, 8-1 thru 8-4, 8-7, Chapter 13
  violation, 6-4, 6-9, 7-7
PUBLIC, 3-2, 3-6, 3-8, 3-9, 10-3, 11-3, 11-6
PUSH, 7-6
PUSHA, 7-7

RCL, RCR, 7-8
readable segment, 2-7
read-only segment, 7-7
real address mode, 10-1, 10-2
real memory management, *see* memory management, real
recovery, *see* exceptions
region, 5-10, 9-4, 9-5
REP, REPE, REPNE, 7-7
requested privilege level (RPL), 2-11, 2-16, 7-2, 13-2 thru 13-4
RESERVE, 3-6, 11-6
reserved word (of descriptor), 2-3, 2-22
RESET, 10-1, 10-3
RESTORE$GLOBAL$TABLE, 2-15
RESTORE$INTERRUPT$TABLE, 2-15
RESTORE$REAL$STATUS, 12-4
return pointer, 6-2, Chapter 7, 12-4, 13-3
return link, *see* return pointer
return address, *see* return pointer
relocation (of segment), 2-4, 2-20, 5-2, 5-3, 5-12, 8-3
RET, 2-7, 2-9, 2-10, 4-3, 11-5
RETURN, 11-5
ROM, 10-1
RPL, *see* requested privilege level

SAVE$GLOBAL$TABLE, 2-15
SAVE$INTERRUPT$TABLE, 2-15
SAVE$REAL$STATUS, 12-4
SBB, 7-8
SCAS, 7-7
scheduling, 2-19, 4-11, 5-12, 9-5
  hardware, *see* scheduling, interrupt
  interrupt, 4-6, 4-7, 4-9, 6-1, 6-5, 6-6, 8-7
  queues, 4-12, 4-13, 9-4, 11-10, 11-14
  software, 4-7, 4-9, 6-1, 6-5, 6-6, 7-6
  state, 4-5 thru 4-7, 4-10
SEGMENT, 11-4, 11-6
segment, *see* logical segment, physical segment
segment limit, *see* limit field
segmented architecture, 9-6
SEGMENT$READABLE, SEGMENT$WRITABLE, 13-2
selector, 2-1, 2-2, 2-8, 2-9, 2-15, 2-16, 3-6, 4-1, 4-3, 5-7, 7-6, 7-7, 11-12, 13-1
  parameters, 2-16, 13-1 thru 13-4
  null, 2-15, 2-17, 7-7
SELECTOR$OF, 3-6
semaphore, 5-6, 5-7, 5-10, 5-16, 9-4, 9-5, 11-2, 12-4, 13-1, 13-4, 13-5
send privilege level (SPL), 13-5
SGDT, 2-15
shadow task, 6-8, 6-9
sharing (of segments), 2-14, 2-15, 2-19, 2-20, 2-22, 4-4, Chapter 5, 8-7, 9-5, 11-2, 11-4, 13-1, 13-4, 13-5
shutdown, 7-5, 10-2
SI register, 7-7
SIDT, 2-15, 6-2
signal, 4-5, 5-7, 5-10, Chapter 6
single-step flag (TF), 6-2, 7-3
SLDT, 2-14, 9-7
slot, 2-20, 2-22, 2-26, 5-10, 11-6, 11-14, 13-5
SMALL, 11-4
SMSW, 12-1
SP register, 4-2, 4-12, 7-6
SPL, *see* send privilege level
SS register, 4-2, 4-12, 7-5 thru 7-7, 9-2, 9-6, 10-1
stack, 2-5, 2-8, 3-1, 4-12, 6-2, 6-4, 7-1, 7-2, 7-6, 7-7, 9-3, 9-4, 9-7, 10-2, 11-6, 11-9 thru 11-11, 13-4
  initial, 4-2, 4-3, 11-6
  overflow, 7-5, 7-7
  exception, 7-5 thru 7-7, 9-2
static system, 1-7, 1-9, 1-10, 2-3, 2-19, 3-1, 4-4, 6-4, 11-9
STI, 4-6, 4-7, 5-6, 6-2, 8-2
STOS, 7-7
STR, 4-1, 4-7, 4-11